ReduxSTM: Optimizing STM designs for Irregular Applications

Manuel Pedrero, Eladio Gutierrez, Sergio Romero, Oscar Plata

Dept. Computer Architecture, University of Malaga, Spain

Abstract

The exploitation of optimistic concurrency in modern multicore architectures via Transactional Memory (TM) is becoming a mainstream programming paradigm. TM features can be leveraged to provide support for speculative parallel execution of irregular applications, characterized by a lack of knowledge about data dependences at compile-time. This work is focused on software TM (STM) solutions and how they can be adapted and optimized to deal efficiently with irregular memory access patterns, mainly those caused by reduction operations.

With this aim, ReduxSTM is introduced as a specific STM system designed by combining techniques for speculative execution with TM algorithms. ReduxSTM is based on three main design aspects: a transactional commit order mechanism which is available to guarantee sequential semantics when needed; a specific transactional memory primitive defined for expressing commutative and associative operations (reductions) that leverages the underlying TM privatization mechanism to avoid unnecessary transaction aborts caused by reduction memory patterns; and an enhanced conflict resolution mechanism that takes advantage of the two previous features.

Keywords: Software transactional memory, Thread level speculation, Irregular applications, Reduction operations

1. Introduction

Modern commodity computers are composed of one or several multicore processors sharing a global memory. The efficient exploitation of these computing resources requires the programmer to invest a considerable effort in developing parallel software. When decomposing a problem into a number of concurrent tasks, the achievable performance is subject to the amount and type of data and control dependences between them. Parallel programs must be carefully designed to allow concurrent thread execution without data races and minimal memory contention on shared data. Given this trend, a key challenge is to make parallel programming technology easier to use. Transactional Memory (TM) falls in such technology, as it is intended to simplify multithreaded programming. TM has emerged as an alternative to traditional lock-based mechanisms to coordinate concurrent threads [1, 2]. TM provides the concept of transaction to enclose a section of code of a thread, enforcing atomicity and isolation during its execution with other concurrent transactions.

TM has been an active research topic for the last two decades [3, 2]. Recently TM is receiving support from processor manufacturers, as evidenced by the implementation of hardware best-effort solutions in the most recent architectures [4, 5, 6]. Apart from hardware designs (HTM), a plethora of software approaches (STM) has been proposed a while ago. TM algorithms can be more flexible and sophisticated in software but at the expense of suffering from performance penalties due to the need for extensive instrumentation to track all transactional memory accesses and to detect and solve conflicts without the help of hardware support. TM algorithms are usually designed for the common case, that is, short transactions with low contention where the majority of transactional memory operations are reads.

The usual domain of TM is multithreaded code; that is, code that it is already parallel. However, as TM exploits opportunistic concurrency (version management and conflict detection/resolution) it can be leveraged to provide support for speculative parallel execution [7, 8] of legacy code. Speculation permits to execute optimistically sections of code from concurrent threads as data updates are buffered and accesses to shared data are tracked to detect conflicts, acting accordingly in such cases to preserve correctness. Programmers can parallelize a sequential application by properly mapping the computations across a number of threads, executing data-dependent code sections as transactions. The underlying TM system dynamically detects data conflicts (dependences) between concurrent transactions and solves them appropriately. In addition, some ordering constraints amongst transaction commits must usually be enforced in order to ensure correct results (preserve sequential semantics).

Speculation is particularly useful in applications exhibiting non-uniform or irregular memory access patterns where dependence information is not easily analyzable at compile time and frequently cannot be solved before runtime. This paper focuses on the challenge of parallelizing irregular applications using

Email addresses: mpedrero@uma.es (Manuel Pedrero), eladio@uma.es (Eladio Gutierrez), sromero@uma.es (Sergio Romero), oplata@uma.es (Oscar Plata)

the speculative support provided by TM. This application class, found in many scientific and engineering domains, contains loops with indirection arrays. This code pattern comes from the fact that data is usually organized as (static or dynamic) unstructured grids or meshes (or compressed sparse matrices). Loops iterate over edges, that represent relations or interactions, and node data is accessed and updated using indirection arrays. Frequently, these loops involve irregular reductions, where array elements are updated using only associative and commutative operations and there are no loop-carried dependences except on elements of the updated arrays.

In this paper we present an approach, called *ReduxSTM*, that optimizes TM mechanisms to support efficient transactions with irregular memory access patterns. The approach is based on combining techniques for speculative execution with software TM algorithms, with the aim of extracting parallelism from irregular applications using transactions. Research focused on combining or exploiting thread-level speculation (TLS) using TM approaches can be found in the literature [7, 9, 10, 11, 12, 13, 14, 15]. However, all these techniques are of general use, not specifically tailored for a particular class of applications. ReduxSTM, in contrast, is designed to extract parallelism from concurrent transactions by exploiting both, commit ordering constraints (in order to preserve sequential semantics) and the ability to privatize irregular reduction patterns. The goal is to avoid unnecessary transaction aborts and rollbacks in the presence of such data access patterns, very common in the broad class of irregular applications. Both features are independent, so if commit ordering is disabled (when the sequential semantics is not violated), ReduxSTM behaves similarly to conventional STMs (not ordered) but with better conflict avoidance in the presence of irregular reduction patterns.

To summarize, this paper makes the following contributions:

- A software TM system, ReduxSTM, that combines speculative execution techniques and transactional memory algorithms to extract parallelism from irregular applications using transactions.
- Transaction commit ordering and irregular reduction patterns are exploited to avoid unnecessary transaction aborts.
- Explicit memory operation primitive and memory set are introduced in the context of TM to optimize the execution of concurrent transactions involving irregular reductions.
- An enhanced conflict resolution mechanism based on additional information coming from ordered transactions.

The rest of the paper is organized as follows. Section 2 introduces irregular reductions and the main approaches designed for their efficient parallelization, in particular those based on transactions. Section 3 discusses an STM model that establishes the basis for ReduxSTM. Section 4 describes ReduxSTM design and algorithms. Section 5 shows an evaluation of the performance and sensitivity of ReduxSTM with various synthetic and application-based benchmarks. Section 6 presents related work. Finally, section 7 concludes the paper.

```
float A;
for (i=0; i<N; i++){
   Calculate ξ;
   A = A ⊕ ξ;
}
```

```
int f1[fDim], ..., fn[fDim];
float A[ADim];
for (i=0; i<fDim; i++){
   Calculate ξ1,ξ2,...,ξn;
   A[f1[i]] = A[f1[i]] ⊕ ξ1;
                ...
   A[fn[i]] = A[fn[i]] ⊕ ξn;
}
```

Figure 1: Examples of reduction loops: single scalar reduction statement (left), multiple irregular reduction statements (right).

2. Transactional approaches to irregular reductions

A reduction statement is a pattern of the form $O = O \oplus \xi$, where $\oplus$ is a commutative and associative operator applied to the memory object O, and ξ is an expression computed using objects different from O. A reduction loop includes one or several reduction statements with the same or different memory objects but with the same operator for each object, in such a way that the only true dependences are those loop-carried dependences due to the reduction statements. Necessarily, in these cases, no references to the reduction objects can appear in other parts of the loop outside the reduction statements [16]. The iterations of a reduction loop can be arbitrarily reordered without perverting the final result, as a consequence of the commutativity and associativity of the reduction operator. Fig. 1 shows examples of reduction loops: a reduction operator $\oplus$ ($+$, $\times$, $max()$...) is applied to a scalar variable A (scalar reduction) or the elements of a reduction array A[] (histogram reduction) [17]. In this last case, the irregular nature of the operation comes from the accesses through the indirection arrays, f1 ... fn, acting as subscripts of the reduction array, which, in turn, are subscripted by the loop index. A great effort has been done in the efficient parallelization of reductions as they are found in the core of many applications and they are frequently associated with irregular access patterns.

2.1. Classical techniques for parallelizing reduction loops

Reduction loops can be executed in parallel as iterations can be reordered thanks to the commutative and associative properties, even if, in general, memory conflicts caused by indirect accesses cannot be detected until run–time. Three groups of strategies are found in the parallelization of reductions: based on mutual exclusion, based on the privatization of the reduction variables and based on the partitioning of the reduction objects. These techniques have proven to be very efficient, although in certain problems it is not easy to anticipate which technique performs the best [18, 19] due to the influence of the memory patterns and other architectural factors such as the synchronization costs.

The first group involves a low programming effort as the loop can be executed in parallel enclosing the accesses to the reduction array in a critical section. Drawbacks of these techniques are the degree of serialization and the cost of synchronization in typical multicore processors. The degree of serialization is basically determined by the number of conflicting iterations and by the particular implementation. In spite of the low programming

effort, a wide range of implementations is available (pure critical section, fine-grained locks, hardware atomic ops, ...) and by choosing one of them we are determining the performance.

The second group of solutions distributes the iteration space into threads, each of which performs its reductions over a local reduction space. A preamble is necessary in order to initialize the private reduction space to the identity (neutral) element of the reduction operator. Similarly, a final reduction phase must accumulate all private reduction values into the (global) reduction array. Two representative examples in this class are *Replicated Buffer* [20] and *Array Expansion* [21]. Optimizations aimed at reducing the memory overhead have been proposed, such as *Selective Privatization/Reduction Table*[18]. These techniques try to minimize memory footprint by replicating selectively only those elements of the reduction array that are written by several threads, but at the cost of complex implementations.

Techniques in the third group avoid the privatization by partitioning the reduction array. They need an inspection phase that is in charge of determining the computation assigned to each thread. In this group we find methods like LocalWrite [22] and SynchWrite [23].

2.2. Motivating examples

Fig. 2 depicts several examples of interest from the viewpoint of transactional memory. These in Fig. 2(a) correspond to situations where reduction sentences are inside a loop, but the loop is not a fully reduction loop. The whole set of iterations cannot be freely reordered, due to reads and writes outside the reduction sentences, or because there is a mixing of different reduction operators. Nevertheless depending on the contents of indirections, a subset of iterations could form a reduction group if they are consecutive, and consequently, some parallelism could be exploitable [24].

The piece of code in Fig. 2(b) takes part of the program *300.twolf* included in the CPU SPEC2000 benchmark [25]. In this case reduction variables are accessed by dereferencing pointers which can induce potential aliases with other objects of the code that can be read or written outside the reduction assignments [26].

Fig. 2(c) shows the work routine of *Kmeans* included in the STAMP transactional benchmark [27]. The loop is a full reduction loop with several reduction sentences. In this case, transactions have been used as a mean to deal with indirections associated to reductions.

As a result of indirect references, classical techniques for reductions may be not applicable just as they are, despite being very efficient. In these cases, parallelism exploitation will involve run-time speculative solutions [28, 26, 24].

2.3. Reduction Parallelization Using Transactional Memory

Transactional memory can be useful to extract optimistic parallelism in the above situations. TM combines all the features of the classical parallelization techniques for reductions (atomicity, implicit selective privatization) with its speculative nature and also an easy programmability.

Transactions can be used as a straightforward replacement of critical sections when parallelizing reductions. After all, a reduction sentence is a read followed by a write in the same memory position, as shown in Fig. 3(a) for an explicit TM system. The execution model is similar to use fine-grain locks protecting individual memory locations [17, 29, 30]. In OpenTM notation [31], Figs. 3(b,c,d) show how the reduction sentences can be enclosed inside transaction with different granularities: assignment by assignment, iteration by iteration or in chunks of iterations. The computation of the values to be added (ξ) is considered private in these examples and it can be outside the critical section. Performance degradation may arise here from the fact that indirections can cause conflicts that may make transactions abort (or stall). The abort cost increases with the size of the transaction, though smaller transactions increase the total starting/end overhead.

If at compile-time it can be assured that no other data dependences exist apart from reductions, the TM system can be arranged in such a way that conflict detection can be disabled for reduction loops, as proposed in [32]. Nevertheless if reduction sentences are found in loops with additional dependences, or the contention over reduction variables is high, the performance of the TM can be at risk. It is precisely these situations which are tackled in this paper.

3. Software transactional memory model

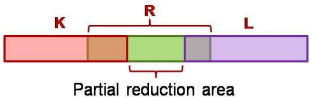
Numerous STM systems have been described in literature. This section tries to summarize some common terminology about STM. A deeper discussion about these terms can be found in [33, 34].

A transaction T is a fragment of a sequential code that has been assigned to a thread which is in charge of its execution. Two or more transactions can run concurrently assigned to different threads. Concerning the memory accesses carried out by a transaction, two definitional properties are assumed: atomicity and isolation [1].

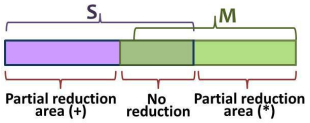
A dataset is defined as the set of memory references associated to a given execution of a transaction T. Typically the set of datasets in a transaction has two members, one for the memory locations that has been written and another for those positions read. We will denote the set of T's datasets as $\mathcal{DS}(T) = \{WS, RS\}$, considering to be formed by the write (WS) and read (RS) sets.

Illusion of isolation and atomicity is achieved by executing transactions speculatively. A transaction T can transit through several possible states, since its assignment to a thread until its end: {*Idle*, *Live*, *Doomed*, *Committing*, *Committed*}. *Idle* state corresponds to those transactions that have not started. *Live* or *Active* means that the transaction is in (speculative) execution and it can be aborted due to data conflicts. In this case we say that the transaction is *Doomed* which is a transitional state that involves to discard speculative changes (rollback) and to retry the transaction from its beginning (making it *Idle* again). When *commit* starts, a first phase can validate possible data conflicts making the transaction abort as well. If correctly validated, during the *Committing* state the speculation ends by consolidating

```
for (i=0; i<N; i++){
   A[K[i]] = ...;
   ... = A[L[i]];
   A[R[i]] = A[R[i]] ⊕ ξ;
}
```




```
for (i=0; i<N; i++){
    A[S[i]] += ξ+;
    A[M[i]] *= ξ*;
}
```



(a) Partial reductions

```
new_dbox_a(..., *costptr) {
  ...
  for (termptr = ... ; termptr = termptr->nextterm) {
    ...
    for (netptr = ... ; netptr=netptr->nterm) {
      ...
      /* Reduction sentence */
      *costptr += ABS(newx - new_mean) - ...;
    }
    ...
    /* Several reads of rowsptr which potentially
     * aliases with the reduction variables */
    rowsptr = tmp_rows[net] ; /* tmp_rows is global */
    for (row = 0 ; rowsptr[row] == 0 ; row++ ){
      ...
    }
    ...
    /* Writes to other global variables which
     * potentially aliases with
     * the reduction variables */
    tmp_num_feeds[net] = f ;
    ...
    tmp_missing_rows[net] = -m ;
    ...
    /* Reduction sentence */
    delta_vert_cost += ( ... );
  }
  return;
}
```

(b) Pointer aliases in a SPEC 2000 code function

```
do {
   ...
   delta = 0.0;
   for (i = 0; i < npoints; i++) {
      index = common_findNearestPoint(
            feature[i], nfeatures,
            clusters, nclusters);
      /* If membership changes,
       * increase delta by 1 */
      if (membership[i] != index) {
         delta += 1.0;
      }
      /* Assign the membership to object i
       * membership[i] can't be changed by
       * other thread */
      membership[i] = index;
      /* Update new cluster centers :
       * sum of objects located within */
      new_centers_len[index][0] += 1;
      for (j = 0; j < nfeatures; j++) {
         new_centers[index][j] +=
                     feature[i][j];
      }
   }
   ...
} while (delta > threshold);
```

(c) Reduction patterns in the STAMP code *Kmeans*

Figure 2: Motivating examples.

```
#pragma omp transaction {
   ...
   A = A ⊕ ξ
}
```

```
TM_start()
   ...
   TM_write(A, TM_read(A) ⊕ ξ)
TM_end()
```

(a) Reduction sentence in an explicit TM system

```
#pragma omp parallel for
for (i=0; i<fDim; i++){
   Calculate ξ1,ξ2,...,ξn
#pragma omp transaction
   A[f1[i]]=A[f1[i]] ⊕ ξ1
#pragma omp transaction
   A[f2[i]]=A[f2[i]] ⊕ ξ2
                ...
}
```

(b) Fine-grain transactions

```
#pragma omp parallel for
for (i=0; i<fDim; i++){
   Calculate ξ1,ξ2,...,ξn
#pragma omp transaction
   {
   A[f1[i]]=A[f1[i]] ⊕ ξ1
   A[f2[i]]=A[f2[i]] ⊕ ξ2
                ...
   }
}
```

(c) Coarse-grain transactions

```
chunk = fDim/nThreads;
#pragma omp transfor
   schedule(static, chunk)
for (i=0; i<fDim; i++){
   Calculate ξ1,ξ2,...,ξn
   A[f1[i]]=A[f1[i]] ⊕ ξ1
   A[f2[i]]=A[f2[i]] ⊕ ξ2
                ...
}
```

(d) Extra-coarse-grain transactions

Figure 3: Parallelization of reduction loops using TM.

the memory positions written by the transaction. After that we say that the transaction is *Committed* and this thread can start a new idle transaction or continue with non-transactional code. Note that in our terminology a re-execution of a given transaction T due to aborts is no considered another distinct transaction but the transition of T to the *Idle* state via *Doomed*.

A set of transactional primitives {*start(T)*, *read(A)*, *write(A, value)*, *abort(T)*, *commit(T)*, *validate(T)*} are available. These operate on a memory address A or over a transaction T. Each primitive can imply the modification of the datasets of the transaction and also may change its state. The *commit()* primitive is in charge of atomically consolidating speculative data into memory. The *abort()* primitive discards all speculative data, making the transaction re-start (rollback) with empty datasets. Notice that *abort()* can be invoked from *read()*, *write()* or *commit()*, if the speculative private data of the transactions is inconsistent with memory data. This validation is done via *validate()* primitive which can check datasets in order to do that or simply can check if the transaction is *Doomed* as a consequence of the conflict detection mechanism.

We say that there is a conflict between two transactions T_1, T_2 if $\exists D_1 \in \mathcal{DS}(T_1), D_2 \in \mathcal{DS}(T_2) \mid D_1 \bigcap D_2 \neq \emptyset$. Potential conflicts are given by $\mathcal{DS} \times \mathcal{DS}$, which are in the typical case the combinations R-R, R-W, W-R, W-W. For each combination a conflict resolution rule must be applied that guarantees memory consistency in the case that the transaction commits [35, 36]. Generally R-R is ignored, but any other pair involving a write will doom one of the contending transactions to abort. Conflict detection may be carried out when a transactional primitive is invoked. Thus several design aspects can be considered. One of them is which transaction is doomed (contention management), finding two major options, to doom the one detecting the conflict (self-abort) or the other one (kill), although another policy can be established [37]. When the conflict is detected for each read or write primitives the conflict detection is said to be eager (encounter time). However, if the detection is deferred to the commit, during the phase before consolidating data, the conflict detection is said to be lazy (commit time).

Concerning the transaction execution correctness, the opacity property [38, 39] is a desirable feature that goes beyond the serialization and guarantees that a transaction aborts before any memory operation makes the memory private snapshot inconsistent with respect to the global memory (because some transactions committed in the meantime). Opacity checking, when

done through incremental invalidation, involves a complexity that grows quadratically with the number of reads, because for each read all previous reads need to be checked for consistency. If done by invalidation in commit-time, this complexity can be reduced to a linear increasing but at the expense of locking memory operations of one transaction while another is committing [40]. As a lighter alternative, *virtual world consistency* [41] behaves similarly to opacity for the committed transactions but it is more relaxed for the aborted transactions. Basically, virtual world consistency is equivalent to working with a consistent fixed snapshot of the memory taken at the start (or restart) of the transaction. Private data are internally consistent during the transaction life but not necessarily with data in the global memory. Virtual world consistency is considered to be enough to tackle typical problems coming from the lack of consistency when opacity is not guaranteed.

Strictly speaking, a transactional system can be considered unordered, as the first transaction that comes to commit, with validated datasets, commits. This is the semantics of transactions as a substitute for critical sections. When used as a tool for parallelizing applications, this unordered behaviour can lead to a valid solution though it is not exactly the same as the one of the sequential code. Or the solution is the same if the sequential code can be safely reordered as it is the case of pure reduction loops. Nevertheless an explicit total order can be introduced. This means that each transaction has an ordinal number which determines the commit order. This order can be defined by the programmer, for example to specify the order of the sequential code. Also, this ordering could be automatically defined from the instant when the transaction starts, or it arrives at commit. Although defining an order could harm performance, it can also play an important role. For example, this relation can be used to set an abort priority ensuring forward progress (no neverending abort cycles). Also certain data conflicts are eliminated as those caused by W-W dependences. Transaction order can also be introduced for semantic reasons, to make the execution equivalent to this one of the sequential (single-thread) execution as it happens in TLS solutions.

Reduction operations involve a read followed by a write in the same memory position and they can be considered as a whole operation in memory. In fact, as discussed previously, concurrent reductions can be implemented as atomic. If we introduce reductions as a new primitive, those conflicts derived from them can be filtered thanks to their commutative and associative nature. Basically transactional reduction primitives will behave leveraging the internal selective privatization associated with the transactional mechanism: (1) Transactional reduction objects are implicitly initialized to the operator neutral element, thereby the first reduction operation initializes the private copy of the reduction variable, (2) subsequent reductions of the same kind will make an incremental reduction, (3) in commit phase, partial cumulated value must be atomically reduced into the global reduction object. The key here is that reduction operations of the same kind do not conflict. So by giving support to reduction operations, it is expected a diminution of the abort rates. Observe that a reduction operation does potentially conflict with other memory operations: reads, writes or reductions

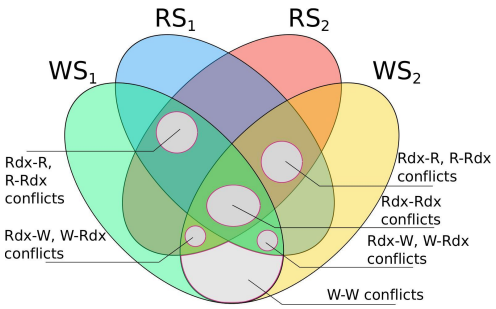


Figure 4: Improvement space considering both support for reductions and ordered transactions. Venn diagram depicts two transactions' data sets.

of another kind. And this can happen both intra-transaction and inter-transactions. Intra-transaction conflicts can be solved using the last private value of the object and may mean to lose the reduction benefits. On the other side, inter-transaction conflicts involving reductions need be solved by settling new conflict resolution rules.

In Fig. 4, the Venn diagram for the write and read set of two transactions is depicted. Note that set size is arbitrary, and this will be in function of the particular datasets in a real case. Marked areas represent the room for improvement when support for reductions is implemented along with a transactional ordering mechanism. By way of illustration only one kind of reduction operator is heeded. Situations with no advantages (nor disadvantages) are the ones where a reduction conflicts with a read (Rdx-R)[1], which remains as a true W-R dependence. Central area corresponds with reduction-reduction (Rdx-Rdx) conflicts, which can be ignored as resolved by the effective privatization carried out by the transaction. In addition, by introducing a commit order, those conflicts caused by modifications in memory (W-W, Rdx-W, W-Rdx) can be also ignored as well as R-W and R-Rdx, because updates are consolidated into memory in transaction order.

4. ReduxSTM

In this section we introduce *ReduxSTM*, a software transactional memory implementation able to take advantage of the commutative and associative properties of the reduction operators in order to reduce the potential transaction aborts carried by reduction variables as discussed above.

4.1. Features

ReduxSTM combines two major features: (1) a specific reduction support by means of new *reduction sets* for each supported reduction operator, which will demand new rules to solve their potential associated conflicts; and (2) an ordering mechanism to set up a commit order for transactions. This last

[1]With X-Y we are denoting two operations X,Y performed by two transactions where the transaction performing operation X precedes the transaction performing operation Y, in the defined order.

feature facilitates conflict resolution as well as permits to preserve the sequential equivalence, if needed, as it happens when TM is used for TLS purposes.

Although *ReduxSTM* could be seen as a general support that can be used to improve a given STM system, some particular characteristics are described next:

Reduction primitives *ReduxSTM* deals with reduction sentences without further knowledge about the surrounding control structure, that may be a reduction loop or not. With this purpose, ReduxSTM defines a new explicit transactional reduction primitive whose arguments are a memory position A, a kind of reduction operation op (sum, product, max, min ...), and a value *deltavalue* to be reduced into A:

redux(A, deltavalue, op) .

The action of *redux()* is functionally equivalent to:

write(A, read(A) $\oplus$ *deltavalue)* ($\oplus$ is the op operator)

with the proviso that different reduction operations of the same kind will not conflict. This primitive is available to the programmer together with the standard *read(A)*, and *write(A, value)* functions, and its use is optional.

Reduction sets Along with the standard Read and Write sets (RS, WS) defined per transaction, a Reduction set needs to be introduced for each reduction operator op (RxS^{op}). When a reduction primitive of operator op is invoked, the memory address of the reduction variable is included into its associated reduction set.

Two particularities should be observed:

(1) That an address remains in its reduction set provided that any other *write()* or *redux()* primitive with different operator does not write in the same address in this transaction. If it is the case, this position must migrate from its reduction set to the write and read sets because the reduction properties are broken in this situation.

(2) Linked to each address in a reduction set, the partial reduction value accumulated till now by this transaction needs to be stored. The first time that the operator is applied on a position the value is simply stored (virtually it is accumulated with the reduction neutral element). However, for subsequent operations of the same kind the stored value must be updated by suitably accumulating it with the *deltavalue* argument of the *redux()* primitive.

Version management ReduxSTM has been devised as a lazy system from the viewpoint of versioning, that is, transaction private values are consolidated into memory during the commit phase. Until then, values are stored in a per-transaction private *redo buffer*. Observe that both the write set and the reduction set contain values that must be consolidated. So, in addition to the conventional write buffer, a reduction buffer is defined. The values associated to the write set are directly written to memory. But those associated to the reduction sets must be accumulated with the corresponding ones in memory according to the reduction operator. Note that a transaction keeps the partial reduced value as if it had been initialized with the neutral element. For reduction variables, transactions are behaving as a implicit selective privatization.

Table 1: Inter-transaction conflicts in different STM designs.

	TM	TM + Order	TM + Reduction	TM + Order + Reduction
R – R	no conflict	no conflict	no conflict	no conflict
R – W	abort	no conflict	abort	no conflict
R – RDX	abort†	abort†	abort	no conflict
W – R	abort	abort	abort	abort
W – W	abort	no conflict	abort	no conflict
W – RDX	abort†	abort†	abort	no conflict
RDX – R	abort†	abort†	abort	abort
RDX – W	abort†	abort†	abort	no conflict
RDX – RDX	abort†	abort†	no conflict	no conflict
RDX – RDX′ ‡	abort†	abort†	abort	abort

†No reduction primitive available. In terms of conflicts, a reduction is implemented as write after read (R-W).

‡RDX, RDX′ refer to reductions of different kind (different operators).

Ordered transaction commits Another key aspect of ReduxSTM is the use of sequenced commits following a numeral ordering. ReduxSTM relies on this mechanism to fulfill the atomicity property. This way, only one transaction can be in committing state. Although disjoint transactions cannot commit concurrently, introducing ordered commits has several advantages in our approach aside from avoiding detecting disjointness. First, it simplifies the contention management as the committing transaction is responsible for killing either other conflicting transactions or itself, depending on the contention policy. Second, if the ordering is defined according to the sequential execution, the result should be identical, which could be desirable in numerical or scientific codes addressed with speculative parallelization. Additionally, as mentioned previously, ordered transactions can enhance the conflict detection mechanism, filtering those conflicts involving a write after a write.

ReduxSTM defines two ways of specifying ordering between transactions. An explicit order can be declared when transaction starts, so the n-th transaction cannot commit until all precedent transactions $1, \ldots n-1$ have committed. For example, this explicit order can be the one of the iterations in a loop. Optionally, an implicit order can be defined, in which transactions obtain their order automatically just before entering its commit phase. This last option resembles an unordered mode, but once the order number is picked up, the transaction keeps it even if the transaction aborts and restarts.

Conflict management As well as the version management, the conflict detection in ReduxSTM is also lazy, i.e., conflicts are not detected until commit. Although lazy conflict de-

tection will make commit probably longer, it allows the system to ignore conflicts that do not imply a read-after-write situation.

Table 1 features those inter-transaction conflicts that do not cause aborts thanks to the combination of the reduction support, ordered commits and lazy versioning and conflict detection. Basically the reduction support eliminates conflicts between reductions of the same kind in different transactions, and the ordered commits rid of these due to writings in the same memory position by two different transactions.

Observe that reductions can give rise to intra-transactions conflicts, when a reduction variable collides with a write or another reduction of different kind. If so, this reduction variable must stop being considering as reduction and must be switched to an standard write, as commented previously. Notice also that our definition of transactional reduction operation does not return the value of the reduction variable. Reductions are associated to updates of memory locations only. In this way, W-RDX does not cause a read-after-write dependence in our implementation (no-conflict).

Contention management The contention management is conditioned by the fact that commits are ordered. In case of conflict between two transactions, the more natural way is to make abort and restart the transaction with the highest order. ReduxSTM is formulated so that the committing transaction is in charge of detecting and solving the conflict situations. Two strategies has been explored:

(1) Kill itself: in this case conflicts have occurred with precedent transactions that have already committed. This is suitable for conflict detection approaches based on timestamps or memory values, as the precedent transactions no longer exist, but their trail is kept as a version number (timestamp) or value.

(2) Kill others: in this case the transactions that may conflict with the committing one, are live or waiting for committing but necessarily with a higher order number. The committing one will mark these conflicting transactions as doomed, which will be aborted/restarted where appropriate.

4.2. Design

ReduxSTM is offered as a library available to the programmer or compiler. It comprises: initialization functions (StmInit( ) to be called by the master thread every time a transactional phase starts in the program and a per-thread initialization function StmInitThread( ) to be called by the thread before starting any transaction), the corresponding closing functions, and the rest of basic primitives defined in an explicit STM as well as the new reduction primitive(s) (StmStart( ), StmRead( ), StmWrite( ), StmRDX( ), StmRollback( ), StmCommit( )). For the sake of a clearer description, one single generic reduction operator is considered hereinafter (represented as $\oplus$), although many others could be considered.

The design of ReduxSTM has tried to keep the essential part of an STM but adding the above mentioned features, in order to evaluate the impact of the support for reductions. This support has been implemented over two different STM approaches that cover the two contention manger strategies commented previously. The first one utilizes the notion of *timestamps*, or version numbers, for conflict detection, which is a popular technique in STM designs (e.g. LSA algorithm [42, 43], but with no locks (see algorithm 1). The second implementation follows a *commit-time invalidation* strategy [40] with a conflict detection based on memory addresses. This is shown in algorithm 2. Table 2 abridges some formal notation used in the description of both algorithms.

Both approaches share the reduction support and the ordered commit features. The support for reductions involves to implement the new primitive StmRDX( ) and modifications to the standard *read* and *commit*, in such a way that the reduction operator(s) is applied when read a reduction value, or when reduction values are consolidated into memory respectively. Information about reduction variables are kept in the reduction set and its associated reduction buffer which stores not the absolute value but the accumulated partial value of the reductions. Observe that a *write* on a previously reduction address involves to migrate this address suitably from the corresponding reduction set/buffer to the write set/buffer, stopping being a reduction.

On the other hand, sequenced ordered commits form the foundation of the transaction synchronization. A global counter, with initial value 1, is incremented when a transaction finishes its commit. The transaction order is defined explicitly when StmStart(order) is invoked. The first action of StmCommit( ) is precisely to wait until the global counter reaches the transaction's order number, in which case the commit will advance if no conflict is detected. Otherwise one of the two transactions need to be aborted. In the timestamp-based approach, the policy is kill-itself and in the commit-time invalidation the policy is kill-others. Notice that only one transaction can execute the commit part beyond the loop waiting for its turn, and this transaction must have the lower order number of all the live ones at this moment.

To guarantee forward progress, the mapping of numbered transactions to threads must be carried out in such a way that the set of live transactions must be a consecutive rank at every moment. If an order number was lost, there would be a transaction waiting for a non-existing one for ever.

Null can be used as a special order number as the argument of StmStart( ). This indicates that the order is implicit, as described in subsection 4.1. In this mode the order is not assigned until the transaction arrives at the commit, just before waiting. With this purpose, another global counter is introduced, which is atomically increased (by fetch-and-add), like a ticket, whenever an implicit order is assigned to a transaction.

The timestamp-based approach in algorithm 1 uses the global order counter as a clock. A global write timestamp array, *globalWTR*, is in charge of storing a timestamp (version number) each time that an address is consolidated in a commit. A

Table 2: Some notation used in algorithms 1 and 2.

$nThreads$	▷ Number of cooperating threads.
$globalOrder$	▷ The global order counter, which is increased when a transaction commits.
$globalTx$	▷ A global array with transaction descriptors, one per thread.
tx	▷ A per-thread global object (thread local storage) pointing to the descriptor of the transaction being executed by such a thread.
$tx.status$	▷ State of a transaction tx. Possible states are: {TxNone, TxIdle, TxActive, TxDoomed, TxCommiting}.
$tx.RS$, $tx.WS$, $tx.RdxS$	▷ Read, Write and Reduction sets of a transaction tx. One Reduction set needs to be defined for each reduction operator. As an example, only a reduction operator is considered ($\oplus$). Write and Reductions sets has a Write and a Reduction buffer associated respectively. The assignment $tx.WS[addr] \leftarrow value$ denotates introducing the *value* for the address *addr* in the WriteBuffer. This assignment involves $tx.WS \leftarrow tx.WS \cup \{addr\}$. Resetting the write set (e.g. $tx.WS \leftarrow \emptyset$) involves to empty also the write buffer. If $a \notin tx.WS$, then $tx.WS[a]$ returns `null`. The same notation applies to reduction sets and buffers.
$globalWTS$, $tx.RTS$	▷ Timestamp arrays: respectively, the global write timestamp and a per-thread read timestamp. The assignments $TS[addr] \leftarrow t$ denotes that the timestamp associated to address *addr* is t. Timestamps are defined from the global order counter. Similar notation to write buffers is used, for resetting and consulting. If a position a has never been written in a commit $globalWTS[a] = 0$. Similarly, for a transaction tx, if $a \notin tx.RS$ then $tx.RTS[a] = 0$.
$*addr$	▷ Contents of the global memory location pointed by *addr*.
savePC&Env( )	▷ Save program counter, stack pointer, and core environment, such as the C function `setjmp()`.
restorePC&Env(env)	▷ Restore program counter, stack pointer, and core environment, such as the C function `longjmp()`.
MemoryFence( )	▷ Full memory fence synchronization primitive, such as the gcc intrinsic `__sync_syncronize()`.

per-transaction read timestamp array, $tx.RTS$, gets the current version number from *globalOrder*, each time that an address is read. Conflicts are tested in commit phase by comparing both arrays, being true when a newer version of a read value exists. In this case, the committing transaction must roll back.

In algorithm 2 the committing transaction has the ability of invalidating subsequent active transactions that have conflict with it. Here no timestamp marks are used, but data set information is used instead. When a transaction reaches its commit phase, and takes its turn, first it updates the shared memory with the private data stored in the write and reduction buffers. Then the commit phase continues checking if another active transaction has one of the updated addresses in its read set. If so, the subsequent conflicting transaction is marked as doomed, as it must abort. Observe that no more than $nThreads - 1$ transactions can be doomed by the committing one.

4.3. Implementation issues

In the design of a software TM system, in addition to the choice of a particular algorithm, much of the performance can come from the implementation details. In this subsection we discuss some implementation details in relation to our proposal.

4.3.1. Synchronization mechanisms

In ReduxSTM, transactions does not use locks to acquire the ownership of memory positions, and in this sense the algorithm can be classified as non-OREC [2]. Instead, combined with the lazy-lazy design, the mechanism that guarantees atomicity is the waiting for the commit turn which is in charge of sequentializing commits in a given order (Fig. 5(a)). In fact, this wait loop is behaving as a spin lock that avoid simultaneous conflicting updates in global memory. This solution matches the premises of subsection 4.1 at the expense of losing certain performance which make it mandatory to keep the commit phase as light as possible.

In modern multicore systems with relaxed memory models some extra synchronization via memory fences/barriers [44] may be necessary. Observe the commit-time invalidation of algorithm 2. Consider a committing transaction simultaneously with a subsequent transaction that is reading. To be correct, these two pairs of actions:

$T_{committing}$: (consolidate values to global memory, check others' ReadSet)

$T_{reading}$: (update my ReadSet, read from memory)

must be seen in this exact order by each thread with respect to the other. In case of conflict, either the checking is done after updating the read set (which causes an abort of the reader transaction) or a correctly committed value is read from global memory though the update of the read set is done prior the checking (there is not any abort but it is unnecessary). All possible execution histories of this two pairs of actions lead to correct situations when each thread perceives them in thread order (see Fig. 5(b)).

Timestamp based version (algorithm 1) does not requires any memory fence, except for consideration about opacity and virtual world consistency, as discussed in subsection 4.3.4, because the committing transaction is validating its data set against precedent already committed transactions (see Fig. 5(c)).

4.3.2. Data structures for write buffers and timestamps

In previous sections write and reduction buffers have been introduced as data structures storing the private speculative values that have to be consolidated into memory if the transaction is validated. Our implementation uses a unified structure for all these buffers, depicted in Fig. 6, which has been called *OpSet*. Basically the structure is a simple hash table implemented as a matrix containing pairs $(address, value)$. As the data granularity used by ReduxSTM is 8 aligned bytes, the 3 least significant bits of the address are not meaningful and they has been used to code whether a value is a result of a write or a reduction. This allows unifying write and reduction buffers in the same structure. Also it facilitates the migration from a reduction buffer to the write buffer simply by changing this 3 address bits.

Operations on the structure include: inserting and finding by address, and traversing all inserted pairs. Each row is selected by applying a hash function to the address. Insertions involve to

Algorithm 1 ReduxSTM – Timestamp based version.

```
1  function StmInit(nThreads)
2      globalOrder ← 1
3      globalTx ← AllocateDescriptors(nThreads)
4      globalWTS ← ∅
5  end function
```

```
1  function StmInitThread( )
2      tx ← globalTx[myThreadId]
3      tx.status ← TxIdle
4  end function
```

```
1  function StmStart(order)
2      tx.RS ← ∅, tx.WS ← ∅, tx.RdxS ← ∅
3      tx.RTS ← ∅
4      tx.order ← order
5      tx.status ← TxActive
6      tx.env ← savePC&Env( )
7  end function
```

```
1  function StmRead(addr)
2      tx.RS ← tx.RS ⋃ {addr}

3      tx.RTS[addr] ← globalOrder

4      if addr ∈ tx.WS then
5          value ← tx.WS[addr]
6      else if addr ∈ tx.RdxS then
7          value ← (∗addr) ⊕ tx.RdxS[addr]
8      else
9          value ← (∗addr)
10     end if
11     return value
12 end function
```

```
1  function StmWrite(addr, value)
2      if addr ∈ tx.RdxS then
3          tx.RdxS ← tx.RdxS − {addr}
4      end if
5      tx.WS[addr] ← value
           (⇒ tx.WS ← tx.WS ⋃ {addr})
6  end function
```

```
1  function StmRDX(addr, deltavalue)
2      if addr ∈ tx.WS then
3          tx.WS[addr] ← deltavalue ⊕ tx.WS[addr]
4      else if addr ∈ tx.RdxS then
5          tx.RdxS[addr] ← deltavalue ⊕ tx.RdxS[addr]
6      else
7          tx.RdxS[addr] ← deltavalue
8          (⇒ tx.RdxS ← tx.RdxS ⋃ {addr})
9      end if
10 end function
```

```
1  function StmRollback( )          ▷ Abort and restart
2      tx.status ← TxIdle
3      tx.RS ← ∅, tx.WS ← ∅, tx.RdxS ← ∅
4      tx.RTS ← ∅
5      tx.status ← TxActive
6      restorePC&Env(tx.env)
7  end function
```

```
1  function StmCommit( )

2      while tx.order ≠ globalOrder do
           ▷ Wait for its turn
3      end while

       ▷ Now committing
4      tx.status ← TxCommitting

       ▷ Validation
5      for all addr ∈ tx.RS do
6          if tx.RTS[addr] ≤ globalWTS[addr] then
7              StmRollback( )
8          end if
9      end for

       ▷ Update global write timestamp
10     for all addr ∈ tx.RdxS ⋃ tx.WS do
11         globalWTS[addr] ← globalOrder + 1
12     end for

       ▷ Consolidate written data into memory
13     for all addr ∈ tx.RdxS do
14         ∗addr ← ∗addr ⊕ tx.Rdx[addr]
15     end for
16     for all addr ∈ tx.WS do
17         ∗addr ← tx.WS[addr]
18     end for

19     tx.status ← TxIdle
20     globalOrder ← globalOrder + 1
21 end function
```

```
1  function StmExitThread( )
       ▷ Descriptor tx free for be reused
       ▷ ReduxSTM thread exit
2      tx.status ← TxNone
3  end function
```

```
1  function StmExit( )
       ▷ ReduxSTM exit
2      DeallocateDescriptors(globalTx)
3  end function
```

reuse an entry if this address was already inserted (it was found this option faster than adding repeated entries like a log).

In each commit the whole write/reduction buffer needs to be traversed. The auxiliary arrays *inserted* and *row_count* are used as indexes for traversing the structure faster. Note that although in algorithms 1 and 2 the consolidation of reductions and writes appear separate, it is implemented as a whole.

Dimensions of this structure have a clear influence in the performance. Ideally, having as many rows as possible reduces the number of hash aliases and so the sequential traversal of a row when searching for an address. Nevertheless, this will increase the reset cost (notice than resetting the structure involve resetting only the *row counter* array and *count* variable), as the structure have to be reset every time a transaction starts or aborts. Another important point is the locality exploitation when accessing to the structure. In order to be more scalable, and to reduce the cache and page misses, the matrix is organized in planes (see Fig. 6). So the access to element $\texttt{DS}[x][y]$ is done as $\texttt{DS}[x/2^p][y][x \mod 2^p]$, where 2^p is the number of entries in the fraction of a row in a plane. In experiments, a plane width of one or two cache lines got the best results.

A similar hashed structure has been used to keep timestamps

Algorithm 2 ReduxSTM – Commit-time invalidation version.

```
1  function StmInit(nThreads)
2      globalOrder ← 1
3      globalTx ← AllocateDescriptors(nThreads)
4      for all t ∈ globalTx do
5          t.status ← TxNone
6      end for
7  end function
```

```
1  function StmInitThread( )
2      tx ← globalTx[myThreadId]
3      tx.status ← TxIdle
4  end function
```

```
1  function StmStart(order)
2      tx.RS ← ∅, tx.WS ← ∅, tx.RdxS ← ∅
3      tx.order ← order
4      tx.status ← TxActive
5      tx.env ← savePC&Env( )
6  end function
```

```
1  function StmRead(addr)
2      tx.RS ← tx.RS ⋃ {addr}
3      MemoryFence( )
4      if addr ∈ tx.WS then
5          value ← tx.WS[addr]
6      else if addr ∈ tx.RdxS then
7          value ← (∗addr) ⊕ tx.RdxS[addr]
8      else
9          value ← (∗addr)
10     end if
11     return value
12 end function
```

```
1  function StmWrite(addr, value)
2      if addr ∈ tx.RdxS then
3          tx.RdxS ← tx.RdxS − {addr}
4      end if
5      tx.WS[addr] ← value (⇒ tx.WS ← tx.WS ⋃ {addr})
6  end function
```

```
1  function StmRDX(addr, deltavalue)
2      if addr ∈ tx.WS then
3          tx.WS[addr] ← deltavalue ⊕ tx.WS[addr]
4      else if addr ∈ tx.RdxS then
5          tx.RdxS[addr] ← deltavalue ⊕ tx.RdxS[addr]
6      else
7          tx.RdxS[addr] ← deltavalue
8          (⇒ tx.RdxS ← tx.RdxS ⋃ {addr})
9      end if
10 end function
```

```
1  function StmKill(otherTx)          ▷ Explicit abort
2      otherTx.status ← TxDoomed
3  end function
```

```
1  function StmRollback( )            ▷ Abort and restart
2      tx.status ← TxIdle
3      tx.RS ← ∅, tx.WS ← ∅, tx.RdxS ← ∅
4      tx.status ← TxActive
5      restorePC&Env(tx.env)
6  end function
```

```
1  function CheckKilled( )
2      if tx.status = TxDoomed then
3          StmRollback( )
4      end if
5  end function
```

```
1  function StmCommit( )
       ▷ Wait for its turn
2      while tx.order ≠ globalOrder do
3          CheckKilled( )
4      end while
5      CheckKilled( )

       ▷ Now committing
6      tx.status ← TxCommitting

       ▷ Consolidation
7      for all addr ∈ tx.RdxS do
8          ∗addr ← ∗addr ⊕ tx.Rdx[addr]
9      end for
10     for all addr ∈ tx.WS do
11         ∗addr ← tx.WS[addr]
12     end for

13     MemoryFence( )

       ▷ Conflict detection, full commit invalidation
14     for all otherTx ∈ globalTx do
15         if otherTx.status = TxActive then
16             if otherTx.RS ⋂ (tx.WS ⋃ tx.RdxS) ≠ ∅ then
17                 StmKill(otherTx)
18             end if
19         end if
20     end for

21     tx.status ← TxIdle
22     globalOrder ← globalOrder + 1
23 end function
```

```
1  function StmExitThread( )
2      tx.status ← TxNone
3  end function
```

```
1  function StmExit( )
2      DeallocateDescriptors(globalTx)
3  end function
```

in algorithm 1 (*globalWTS*, *tx.RTS*). In this case one single value needs to be stored per entry. During the commit phase the structure associated to *tx.RTS* is traversed in order to validate the absence of conflicts. Note that all of the addresses mapped with the same hash value are aliases from the timestamp viewpoint. Consequently false conflicts due to this aliasing may arise, with higher probability for smaller timestamp structures. If both data structures for *OpSet* and timestamps have the same dimensions, the same hash function can be used and the loop updating global write timestamps can be fused with this one consolidating data into memory in algorithm 1.

4.3.3. Data structures for data sets (commit-time invalidation implementation)

Although *OpSet* has membership information about the Write and Reduction sets, in the commit-time invalidation version of ReduxSTM, it can be inefficient to traverse them when checking conflicts. For this reason in this version Bloom fil-

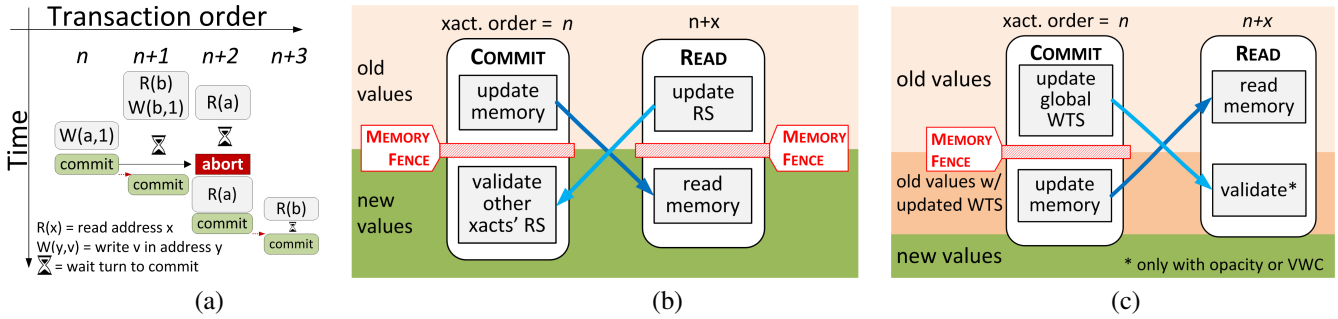


Figure 5: Synchronization mechanisms in ReduxSTM: (a) ordered commits, (b) memory fences in commit-time invalidation algorithm, (c) timestamp based ReduxSTM with opacity.

ters [45, 46, 47] have been introduced. Each transaction has one for representing $tx.RS$ and another one for $tx.WS \cup tx.RdxS$.

Bloom filters have been implemented as unsigned integer arrays, where each bit represents a set of memory addresses mapped by a single hash function. An insertion to the corresponding filter needs to be done for each read, write and reduction operation. Nevertheless, the test for disjointness is very fast as it involves only the bitwise-and of the words compounding the filter.

The performance of Bloom filters is a trade-off of its size. For small filters the probability of false positives may be high, but for large ones the time spent in reset and the traversal to check their intersection can be dominant. Paradoxically, for highly contented transactions, where filters may be very populated, conflicts can be found faster than for almost empty filters as the traversal is stopped when a positive is found.

4.3.4. The opacity issue

Algorithms discussed so far do not verify opacity [38] or another similar condition as we have focused on introducing the support for reductions. Although such correctness conditions require additional synchronization effort at the expense of performance, they can be necessary for the right execution of some particular codes.

Algorithm 3 shows how to incorporate the checking for opacity to the timestamp version of ReduxSTM. Just before the read function returns, a call to a partial validation function is inserted (anticipating the validation done in commit). Two alternatives

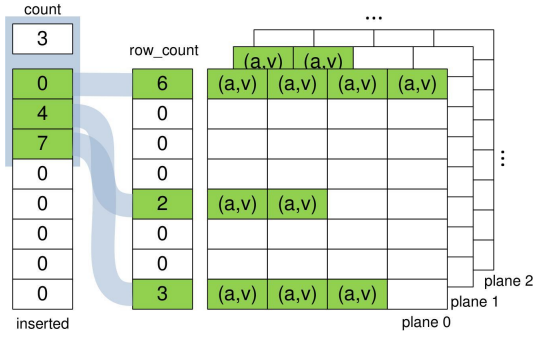


Figure 6: Unified hashed data structure for Write and Reduction buffers (OpSet).

for this validation are featured. ValidateVWC( ) checks for virtual world consistency [41] based on the value of the global order at transaction's start. ValidateFullOpacity( ) checks for opacity by testing all the positions previously read by the transaction. As discussed in section 3, virtual world consistency is a lighter alternative to opacity being enough for a wide range of situations. Observe that a memory fence is needed to guarantee that reading transactions see the correct order of the committing one (Fig.5(c)).

In the commit-time invalidation version a different approach is needed to fulfill opacity because it would be cost prohibitive if transactions have to access each other's data sets (without race conditions). In this case the idea is treating the commit phase as a critical section with respect to reads [40]. To this effect a sequential lock (seqlock) [48, 49] mechanism has been used, as shown in algorithm 4. By means of the global variable *globalOpacitySeqLock*, no reader can start if a commit is in progress, and a read must be retried if a commit was completed during the read routine before returning.

4.3.5. Further optimizations

Our ReduxSTM implementation has taken into account some other optimization issues that include:

- The use of the optimized library *asmlib* [50] for resetting data structures.

  Note that an influencial design parameter is the size of data structures (Bloom filters, timestamp tables, write buffers) because as they are smaller, performance may degrade due to hash aliases in memory addresses. However, in the opposite side, the initialization of large data structures is a frequent operation that can cause also inefficiencies. For this reason, the use of an optimized library to perform operations like `memset()`, used for reset, can be a critical point.

- Irrevocable retry of the committing transaction in the timestamp approach.

  As the committing transaction in this version can kill itself only once, the retry can be done in a irrevocable way [51]. In this manner, the check for conflicts can be removed in this re-execution. Note that if opacity or VWC are en-

Algorithm 3 Opacity or virtual world consistency in ReduxSTM - Timestamp based version.

```
1  function StmStart(order)
2      ...
3      tx.start ← globalOrder
4  end function
```

```
1  function ValidateVWC(addr)
2      if tx.start < globalWTS[addr] then
3          StmRollback( )
4      end if
5  end function
```

```
1  function ValidateFullOpacity(addr)
2      for all addr ∈ tx.RdxS do
3          if tx.RTS[addr] < globalWTS[addr] then
4              StmRollback( )
5          end if
6      end for
7  end function
```

```
1  function StmRead(addr)
2      tx.RS ← tx.RS ⋃ {addr}
3      tx.RTS[addr] ← globalOrder
4      if addr ∈ tx.WS then
5          value ← tx.WS[addr]
6      else if addr ∈ tx.RdxS then
7          value ← (*addr) ⊕ tx.RdxS[addr]
8      else
9          value ← (*addr)
10     end if

       ▷ Call one of the validate() functions
11     Validate(addr)

12     return value
13 end function
```

```
1  function StmCommit( )
2      while tx.order ≠ globalOrder do
           ▷ Wait for its turn
3      end while

       ▷ Now committing
4      tx.status ← TxCommitting

       ▷ Validation
5      for all addr ∈ tx.RS do
6          if tx.RTS[addr] ≤ globalWTS[addr] then
7              StmRollback( )
8          end if
9      end for

       ▷ Update global write timestamp
10     for all addr ∈ tx.RdxS ⋃ tx.WS do
11         globalWTS[addr] ← globalOrder + 1 ??
12     end for

13     MemoryFence( )

       ▷ Consolidate written data into memory
14     for all addr ∈ tx.RdxS do
15         *addr ← *addr ⊕ tx.Rdx[addr]
16     end for
17     for all addr ∈ tx.WS do
18         *addr ← tx.WS[addr]
19     end for

20     tx.status ← TxIdle
21     globalOrder ← globalOrder + 1
22 end function
```

abled, an extra memory fence will be necessary each time a write update its corresponding global write timestamp.

- Anticipated rollback in the commit-time invalidation approach.

  Any live transaction with higher order number than the committing one can be marked as doomed in case of conflict. If any transactional operation (read, write, reduction) checks for this status each time it is invoked we can anticipate the abort. This check introduces a light overhead, but can be advantageous for large transactions.

  It should be mentioned that now it is possible for a commit to check a read Bloom filter of other transaction while such a filter is being reset. In this case a false abort can be raised. However it has been tested that the probability of this situation is very low and it is not worth introducing an extra synchronization to prevent it.

5. Evaluation

ReduxSTM's performance has been tested over a variety of codes summarized in table 3. It covers several groups of scenarios that we have considered of interest. Our goal is measuring the impact of the transactional reduction support as well as analyzing the behaviour of our proposal in general contexts (with no reductions).

Full reduction loops are not sensitive to order (due to commutativity) but from the viewpoint of dependences, conflicts occur if reductions are implemented as reads and writes. This aspect is analyzed with RXasRW. Two of the benchmarks, EigenBench [52] and WormBench [53], have been modified in order to introduce reduction operations which are not present originally. In the modified EigenBench, reduction sentences coexist with reads and writes. Although strictly speaking a correct result would require sequential ordering, we will consider this code as unordered, keeping its original spirit. Our version of WormBench forces worm objects to carry out reduction operations only. This way the code becomes equivalent to a full reduction loop, and consequently unordered and with no partial reductions. Patterns found in CHARMM [54] are representative of full reduction loops, where loop-carried dependences come only from reduction operations on arrays accessed through indirect subscripts. In Twolf, writes, reads and reductions coexist together, but a correct result is guaranteed only if sequential order is preserved. The STAMP suite is a general TM benchmark, frequently used as a reference, but not specially focused on reduction patterns, present only in one of its codes. Finally, from

Algorithm 4 Opacity in commit-time invalidation based version of ReduxSTM.

```
1  function StmInit(nThreads)
       ...
2      globalOpacitySeqLock ← 0
3  end function
```

```
1  function StmRead(addr)
2      tx.RS ← tx.RS ⋃ {addr}
3      re_read:
4        do
5            start ← globalOpacitySeqLock
6        while start mod 2
7      MemoryFence( )
8      if addr ∈ tx.WS then
9          value ← tx.WS[addr]
10     else if addr ∈ tx.RdxS then
11         value ← (*addr) ⊕ tx.RdxS[addr]
12     else
13         value ← (*addr)
14     end if
15     if start ≠ globalOpacitySeqLock then
16         goto re_read
17     end if
18     return value
19 end function
```

```
1  function StmCommit( )
2      while tx.order ≠ globalOrder do
3          CheckKilled( )              ▷ Wait for its turn
4      end while
5      CheckKilled( )
       ▷ Now committing
6      tx.status ← TxCommitting
7      Fetch&Add(globalOpacitySeqLock, 1)
       ▷ Consolidation
8      for all addr ∈ tx.RdxS do
9          *addr ← *addr ⊕ tx.Rdx[addr]
10     end for
11     for all addr ∈ tx.WS do
12         *addr ← tx.WS[addr]
13     end for
14     MemoryFence( )
       ▷ Conflict detection, full commit invalidation
15     for all otherTx ∈ globalTx do
16         if otherTx.status = TxActive then
17             if otherTx.RS ⋂ (tx.WS ⋃ tx.RdxS) ≠ ∅ then
18                 StmKill(otherTx)
19             end if
20         end if
21     end for
22     Fetch&Add(globalOpacitySeqLock, 1)
23     tx.status ← TxIdle
24     globalOrder ← globalOrder + 1
25 end function
```

the SPEC2006 benchmark, several loops with interest from the TLS viewpoint have been tested.

As a reference, TinySTM, one popular state-of-the-art STM library, has been considered in two compilations: the standard base configuration of the library (hereinafter referred as TinySTM) and a compilation with ordered commits (module `mod_order` enabled) (referred as TinySTM-ordered).

Experiments were performed on a 2.30GHz Intel Xeon E5-2698 (Haswell) server with 16 cores and 256GB RAM. All executables were compiled with `gcc` 4.8.2 with `-O2` optimizations.

It should be mentioned that although our server processors support best-effort HTM, so as it would be possible to use it in ReduxSTM in a similar way as in [55], however this was not accomplished. First, because this would need to know the reduction set a priori (for instance, using explicit annotations). Second, this would be worth it only for very small reduction objects, due to hardware capacity overflows. In fact, this is a drawback identified in [55].

The observed execution time has been obtained getting the minimum time of at least 10 trials per experiment. Those tests that exceeded a reasonably execution time (about 10 times greater than the sequential one) were considered timed-out. This situation happened specially with the ordered version of TinySTM. Both proposed implementations of ReduxSTM have been tested: Timestamp based version (ReduxSTM-TS) and Commit-time invalidation version (ReduxSTM-CTI).

5.1. RXasRW (Reductions as Read+Write)

RXasRW is a synthetic histogram reduction loop shown in Fig. 7 whose goal is to measure the impact of the transactional reduction support in terms of the fraction of reduction sentences that benefit from it. A reduction operation on an array is carried out per iteration through an indirection subscript which is initialized with random values. Configurable parameters includes the reduction array size (*NR*) which acts as a measure of contention, the transaction size (iterations per transaction), and the computational load associated to the value to be reduced (ξ).

RXasRW also features a threshold, θ, which determines the probability of the reduction being implemented in the standard way, that is, as a read followed by a write. For a STM supporting reductions, θ=0% means that all reductions are implemented with the special primitive StmRedux, whereas a 100% implies that every reduction is implemented as a read and a write. For other STMs, all reduction operations are carried out in the standard way, independently of θ.

Fig. 9 shows the observed performance in terms of speedup with respect to the sequential execution and TCR (Transactional Commit Rate, defined as the quotient between the number of committed transactions and the total number of them, committed and aborted). These experiments were performed using a loop of 10^7 iterations, a reduction array of 1000 elements, 10 iterations per transaction, and a computational load of 50 FLOPs per iteration.

Table 3: Benchmarks used for evaluation.

Benchmark name	Description	Histogram loop	Partial reductions	Ordered
RXasRW	Histogram loop with a fraction of reductions implemented as Read+Write (see Fig. 7)	yes	yes	no
EigenBench	The TM benchmark Eigenbench [52], suitably modified to include reductions	no†	yes†	no
Twolf	Function `new_dbox_a()` from the code 300.twolf (SPEC2000 CPU) [25]	no	yes	yes
Charmm	Non-bonded force calculation loop of a molecular dynamics 3D simulation kernel [54]	yes	no	no
Wormbench	Yet another STM benchmark [53] modified to include reductions	yes†	no†	no
STAMP	All codes of the generic transactional STAMP suite [27]	no‡	no‡	no
SPEC2006	Selected loops suitable for thread-level speculation of the SPEC CPU2006 suite [14]	no	no	yes

† This refers to our customized version ‡ Except KMeans

```
uint64_t A[NR];   /* Reduction array */
int Ind[N/xactSize][xactSize]; /* Indirection array */

/* Reduction (A) and indirection (Ind) arrays
   has been initilialized previously */

/* Iteration space (N) is assumed to be
   multiple of the transaction size (xactSize) */

for (i=0; i<N/xactSize; i++){
    StmStart();
    for (ii=0; ii<xactSize; ii++){
        Compute ξ
        idx = Ind[i][ii];
        prw = rand();
        if (prw > θ)
           StmRedux(A[idx], ξ);
        else
           StmWrite(A[idx], StmRead(A[idx]) ⊕ ξ );
    }
    StmCommit();
}
```

Figure 7: *RXasRW* pseudocode, a histogram reduction loop where a fraction of reductions are implemented w/o reduction support.

```
void test_core(tid, loops, Writes, Reads, Redux, ...) {
  ...

  for (i=0; i<loops; i++) {
    StmStart();
    for (j=0; j< Writes + Reads + Redux ; j++) {
      switch(rand_action(...) {
         index = rand_index(tid, ...);
        case READ:
          val += StmRead(array[index]);
        case WRITE:
          StmWrite(array[index], val);
        case REDUX:
          StmRedux(array[index], val);
        ...

      }
    }
    StmCommit();
    val += local_ops(R_OUT, W_OUT, val, tid);
  }
}
```

Figure 8: Sketch of the Eigenbench kernel, modified to include reduction operations in addition to reads and writes.

Several facts can be highlighted. First, the speedup and TCR reached by ReduxSTM is appreciably better than this one achieved by TinySTM (both standard and ordered), even though ReduxSTM makes transactions commit in sequential order. Even more, TinySTM underperforms quickly when more threads are competing. This endorses the ability of the proposed support for reductions to filter a considerable amount of conflicts caused by reduction sentences. Second, ReduxSTM-TS is able to scale better for higher number of threads with regard to ReduxSTM-CTI. The reason for that is that the kill-others policy of CTI version leads the committing transaction to check a increasing number of Bloom filters corresponding to the other potentially conflicting threads. Relative discrepancies between TCR and speedups can be explained by the cost of commit phase in ReduxSTM. In this way, although ReduxSTM-CTI and ReduxSTM-TS exhibit similar TCR, the speedup can be lightly different. Third, the partial reductions, coming up from implementing a fraction of reductions as read plus write, causes that the advantage of ReduxSTM grows speedily with the number of reduction sentences implemented via StmRedux (lower θ).

Starting from the initial configuration described above, several sensitivity measures are shown in Fig. 10. All these experiments were executed with θ=0%, that is, maximum TCR for ReduxSTM (all conflicts filtered by the reduction support).

The reduction array size determines the contention pressure (Fig. 10(a)). By increasing this size, the contention decreases. It must be kept in mind that a high contention affects not only the parallel transactioned code but also it slows down the sequential one because locality exploitation may be degraded. Also the cost of instrumentation may be higher for lower contention because a larger range of addresses needs to be tracked (more queries and insertions in data structures). This is specially noticeable for ReduxSTM-TS because commit validation phase depends on the number of unique addresses in data sets. It is precisely in high contended scenarios where ReduxSTM takes a stronger advantage over the reference STM. When the contention pressure decays, this advantage shrinks lightly with respect to the standard TinySTM, although it continues high with regards to its ordered counterpart (remember that ReduxSTM always preserves the order). As stated before, ReduxSTM-TS is able to scale better with the number of threads because it does not need to query living transactions of other threads.

Transaction size must trade the overhead associated to start and finish transactions off the cost of aborts (Fig. 10(b)). Larger transactions involve higher conflict probability and also to discard more work. For the current problem dimensions, this tradeoff is observed more clearly for 8 threads. TinySTM is underperforming deeply in all these experiments, which causes a relative insensitivity to the transaction size, specially TinySTM-ordered. It should be noted that many experiments of TinySTM timed out. Observe again that ReduxSTM-TS scores over ReduxSTM-CTI for a higher number of threads.

Computational intensity (ratio between arithmetic and mem-

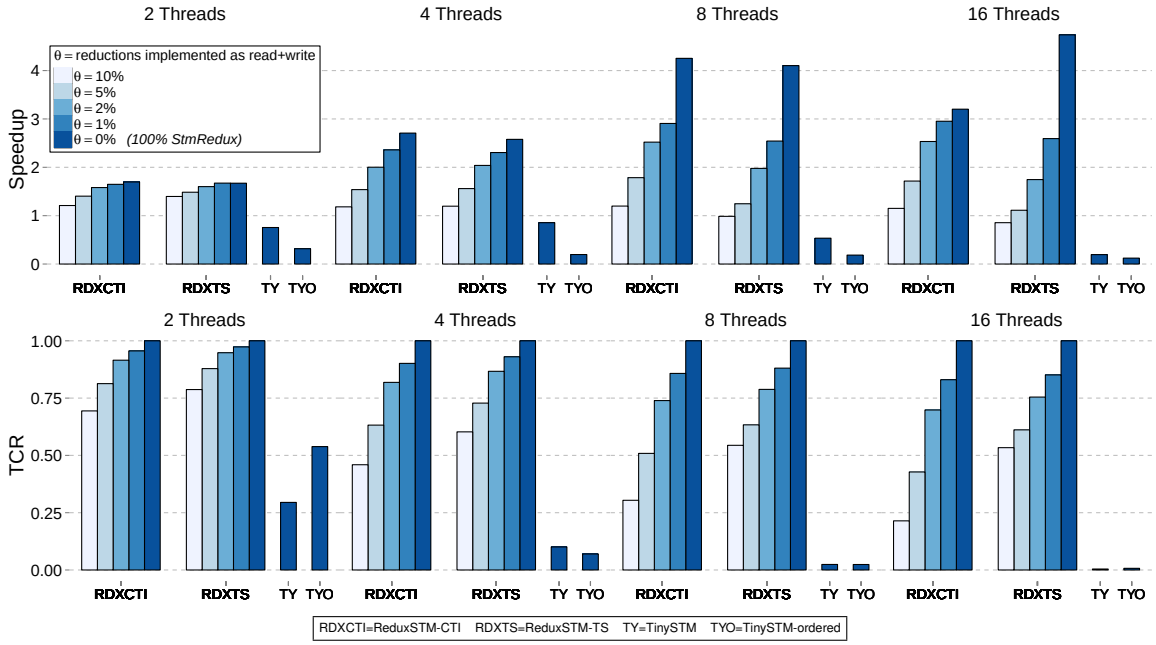


Figure 9: Performance of *RXasRW* in terms of speedup (upper) and transaction commit rate (TCR) (lower).

ory operations) can also modulate the behaviour of the benchmark. This intensity is commonly associated to the computation of values to be reduced into the reduction objects. Although higher computational effort increases the abort recovery cost, in general, it translates into more exploitable parallelism, as shown in Fig. 10(c). Whereas the high conflict rates of TinySTM harm its performance, ReduxSTM specially benefits from high computational loads in scenarios like this where reduction operations predominate.

The need to represent unbounded data sets in an efficient way, has involved to use hashed data structures (Bloom filters in ReduxSTM-CTI, and timestamp arrays in ReduxSTM-TS) whose finite size gives rise to certain probability of conflicts due to false positives. Plots in Fig. 10(d) evidence this effect (θ=3% is considered). A performance optimum can be measured around a size of 2^{10} elements for the analyzed problem size (in both Bloom filters and timestamp arrays), with negligible TCR improvement beyond this point. This is because increasing the hashed data structure size adds an extra overhead (resetting, inserting, querying, ...) that may not compensate the drops in false positives. It should be mentioned that these extra costs are dominated by the resetting of data structures in the case of ReduxSTM-TS, and the intersection of Bloom filters in the case of ReduxSTM-CTI.

5.2. Eigenbench

Eigenbench [52] is a highly customizable benchmark with the possibility of isolating orthogonal aspects that determine the performance of a STM. Original code has been modified to introduce reduction sentences as shown in Fig. 8. While ReduxSTM can leverage its reduction support, other STMs will use the read+write equivalent for these sentences. Experiments have been set up using the so-called *hot-array*, which models a shared array where all threads can access. The size of the *hot-array* has been 100K elements, all transactions in each experiment are of the same size.

Fig. 11 displays several ternary plots sampling the space of memory operations (Read/Write/Reduction) for different transaction sizes. Markers indicate the STM with the highest observed speedup in each sampled scenario. Speedup is computed with the minimum execution time of at least ten repetitions of each experiment.

In short transactions (10 memory operations per transaction) TinySTM obtains the best result except when the number of reads is almost null, in which case ReduxSTM-CTI works better. As conflict probability increases (larger transactions) ReduxSTM-TS starts dominating specially in scenarios with a high number of memory updates, relegating TinySTM to those samples with more reads (from 40 ops./xact. TinySTM is the best only for read-only transactions, while TinySTM-ordered did not get the best result in any experiment).

5.3. TWolf

Function `new_dbox_a()` is part of the code *300.twolf* included in the SPEC2000. Its interest lies in the loop sketched in Fig. 2(b) that contains sentences with potential reduction patterns. However, as shared reduction variables are accessed via pointers, reference aliases may exist. This fact that cannot be determined until execution time [26].

In these experiments, a two-step methodology was used for the evaluation. First, the original sequential program was instrumented to obtain a memory trace of the function of interest (`new_dbox_a()`), marking in it if a position corresponds to a read, a write or a reduction. In a second step, the trace is simulated recreating the instrumented memory pattern, as well as the original computational load. The simulation is carried out in

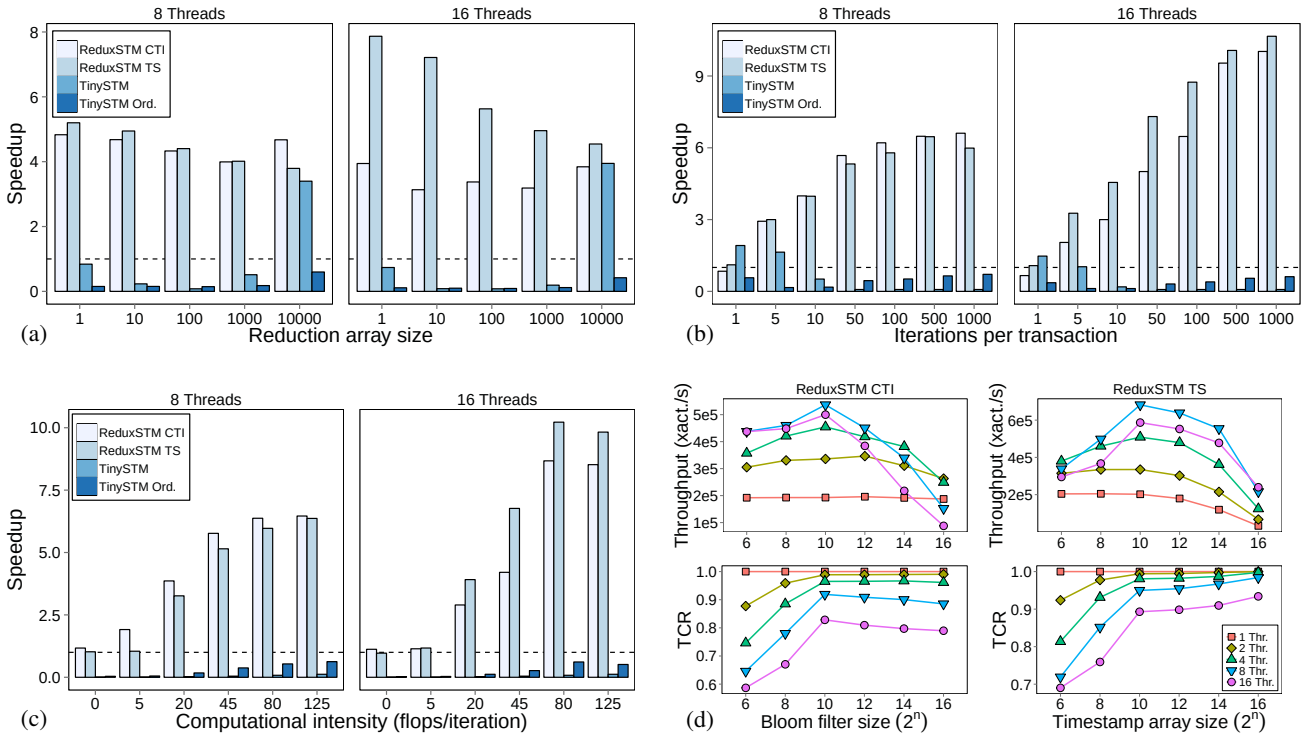


Figure 10: Sensitivity to different configurations of *RXasRW*.

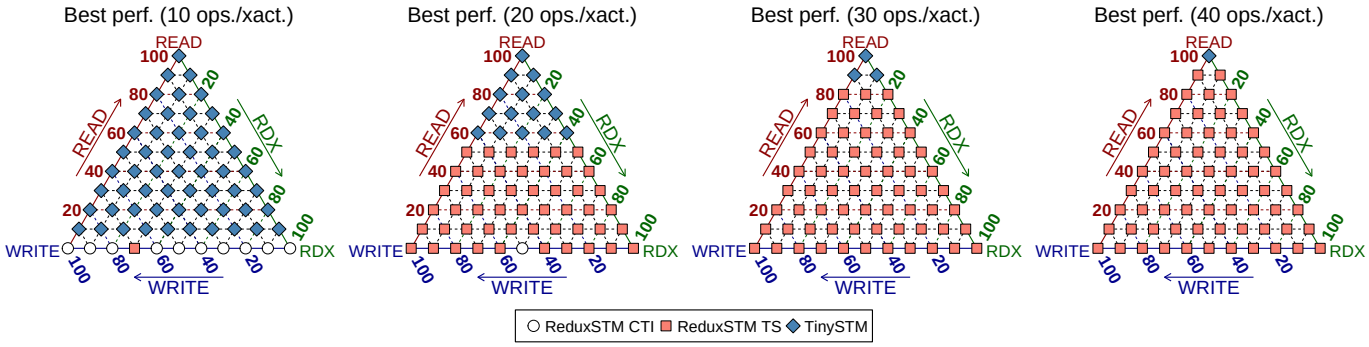


Figure 11: Eigenbench: STM with better performance for a given percentage of reads, writes and reductions (4 threads considered).

parallel and in a transactional way by partitioning the outer loop of the simulated function into chunks of consecutive iterations. Each one of these chunks is executed as a transaction. Transactions are mapped to threads in a round-robin way, preserving the original serial order.

Experiments were carried out using a medium-sized workload (the *training workload* included in 300.twolf). For this load, in total, 12 million iterations of the outer loop in `new_dbox_a()` were executed. The resulting memory trace contained about 550M reads, 46M writes and 70M reductions. An important observation is that less than 0.15% of the total memory references are different. That means a high contented transactional execution. In addition, the amount of exploitable parallelism is limited by the memory-bound nature of the code.

Fig. 12 shows results obtained with different number of iterations enclosed in a single transaction. Although TinySTM (not ordered) has been included for comparison it must be warned that wrong results may be produced in this case as original order is not guaranteed. As a reference, a global lock based parallelization is also shown, which is not able to scale at all due to the highly contended data sets. Observe that best results are obtained for the smallest chunk size (one iteration per transaction). The large number of memory operations per iteration, and the high contention explain why larger chunks harm performance. In contrast to TinySTM, implementations of ReduxSTM get rid of conflicts derived from potentially conflicting reduction patterns. For larger transactions, ReduxSTM-TS performance falls down. ReduxSTM-CTI, while obtains worse

speedups than the timestamp based alternative is shown to be less sensitive to transaction size in the analyzed scenario.

5.4. Charmm

These experiments correspond with the non-bonded force calculation loop described in [54] where each particle of a set interacts with all others. Interacting particle pairs are represented by a neighbour list per particle, which gives rise to reduction patterns with subscripted subscripts and indirections in the innermost loop bounds. Experiments shown in Fig. 13, correspond to a simulation using a system of 2.5K particles involving 372K interaction pairs.

The main observation in Fig. 13 is how the support for reductions helps to solve the underperforming behaviour caused by reductions when they are implemented as transactional write after read, due to absence of conflicts. Even although this loop can be safely reordered (since we have no dependences apart from reduction variables), TinySTM is unable to take advantage of this fact. Another remarkable point is the scalability exhibited by ReduxSTM in the range of number of threads tested.

5.5. Wormbench

Wormbench [53] is designed to evaluate the behaviour of STM implementations and to test some of the common synchronization problems associated with multi-threaded applications. This benchmark simulates several worms moving in a world represented by a shared matrix of values. Each worm has a body and a head, covering a subset of the world, and it is controlled by a single thread. In each step of the simulation, all the worms perform a move, involving some operations in the subset of the world covered by the worm head. The action of the worm over the world must be atomic. Different scenarios can be simulated by configuring the size of both world and worms, and the sequence of operations to be performed in each simulation step.

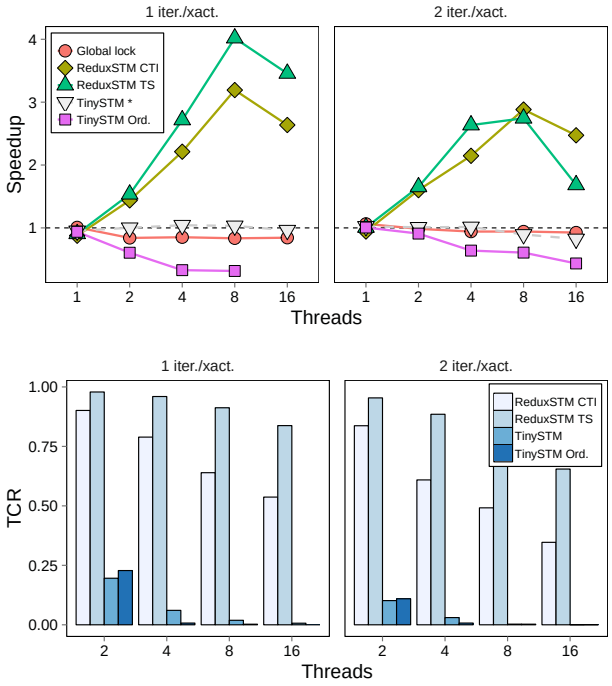


Figure 12: Speedup (left) and TCR (right) for routine *new_dbox_a()* in *300.twolf* code (SPEC CPU2000).

Wormbench has been originally written in C# and for our purposes it has been ported to C language in order to be compiled with the STM libraries under study. A new worm operation has been introduced. This carries out a histogram reduction on the world elements covered by the head of the worm. The experiments has been focused on this new operation in such a way that all data dependences come from reductions (no partial reductions).

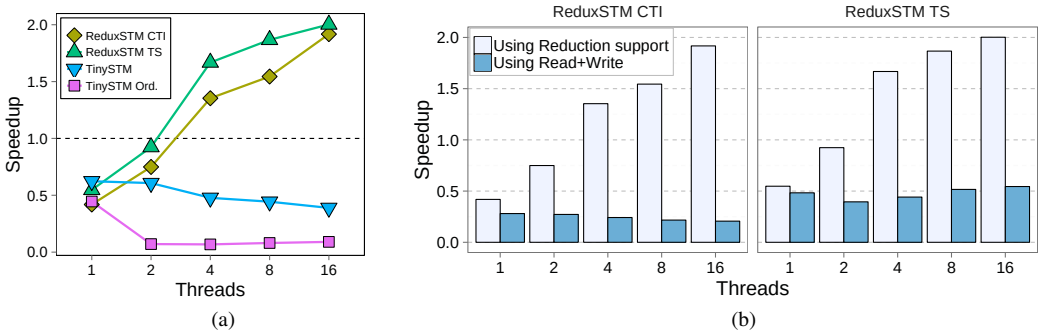
Results shown in Fig. 14 correspond to experiments using a constant head worm size of 8 elements and a range of world sizes from 16x16 to 128x128. World size will determine the contention degree: the smaller the world, the higher the contention, and consequently higher conflict probability. TinySTM versions present difficulties to perform efficiently in high contended scenarios. In fact, TinySTM-ordered exhibits a very poor performance and the standard version times out in small worlds for 16 threads (represented with null speedup in the figure). An interesting fact is that the contention level where ReduxSTM and TinySTM equals performance grows with the number of threads. This point takes place in a 24x24 world for 2 threads but around 64x64 for 16 threads. Observe also that although ReduxSTM-CTI performs slightly better than ReduxSTM-TS for 2 and 4 threads, ReduxSTM-TS outperforms the former for 16 threads.

5.6. STAMP

Stanford's STAMP [27] is a benchmark suite designed for transactional memory that tries to cover a wide spectrum of algorithms and domains. It includes 8 benchmarks based on real applications with several configurations oriented to evaluate TM implementations. Experiments used the default parameters from [27] which are summarized in table 4. It should be emphasized that although this suite is not specially focused on reduction patterns, it can be useful to measure the performance of ReduxSTM in general situations.

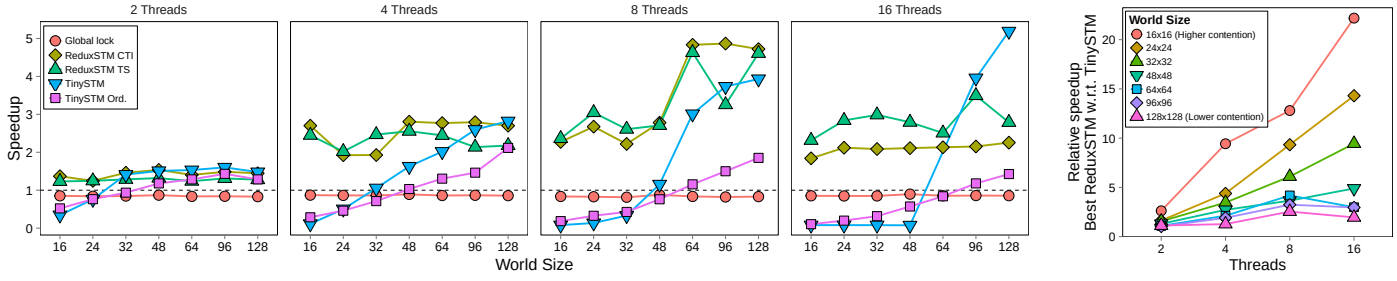
In general, as observed in Fig. 15, ReduxSTM performance is placed between the ordered and unordered version of TinySTM. Reasons for that include: the overhead associated by the ordered commits in ReduxSTM, the low abort rate of the analyzed configuration and the absence of reduction sentences to be exploited. In codes like SSCA2, KMeans and Intruder, the small size of transactions limits the performance of ReduxSTM-CTI. It performs much better with large and contended workloads, like in Bayes or Labyrinth. On the other hand, overall ReduxSTM-TS results are better, even as competitive as TinySTM-ordered in most situations.

It should be mentioned that for the only code containing reduction patterns, KMeans (see Fig. 2(c)), the improvement of ReduxSTM is not as significant as expected. Two facts explain this. First, transactions are enclosing several small code sections with reduction sentences which causes a barrier effect between transactions of different threads in ordered STMs. This



(a) (b)

Figure 13: Charmm; (a) observed speedup (b) ReduxSTM speedup improvement using the support for reduction operations versus implementing the reductions as write after read.



(a) (b)

Figure 14: Wormbench benchmark; (a) sensitivity to different world sizes, the smaller the world, the higher the contention; (b) summarized relative speedup (best ReduxSTM version w.r.t. TinySTM-ordered) for different contention degree.

damages particularly the performance of ReduxSTM. Second, the use of a queue to distribute the work adds some artificial R-Rdx dependences due to a conditional check inside the reduction loop. This diminishes the capacity of ReduxSTM to filter conflicts due to reductions. Notice, that both problems are programming artifacts that could be overtaken if KMeans is recoded, as the algorithm by itself can be considered a histogram reduction loop.

Another remarkable fact is that some of the codes require at least virtual world consistency (VWC) to be executed correctly (Bayes, Yada, Indtruder) [56]. Fig. 16 shows the impact in ReduxSTM-TS performance that relaxing consistency conditions have in those codes where it can be safely done. Serializability obtains the best results in performance as the consistency is more relaxed. At the other extreme opacity offers very strong guarantees, but it demands a higher computational effort. ReduxSTM encounters a good consistency balance with VWC, that guarantees a reasonable consistency degree without worsening too much performance.

5.7. SPEC2006

This subsection includes experiments with some interesting loops from the SPEC CPU2006 benchmark suite. The goal is to illustrate the thread-level speculative capabilities of ReduxSTM, i.e., how ReduxSTM behaves in a more general range of codes with no special emphasis in reductions but suitable for speculative approaches. In this context, the execution correctness is preserved by committing transactions in the equivalent sequential order. The loops have been selected from those analyzed in [14] which show potential speedup using thread-level speculation techniques according to previous studies [57]. Particularly, the chosen loops are: `pbeampp.c:165`, `quark_stuff.c:1523`, `fast_algorithm.c:133`, `mv-search.c:394` and `vector.c:513` from benchmarks 429.mcf, 433.milc, 456.hmmer, 464.h264ref and 482.sphinx3, respectively. They represent a significant portion of the execution time of each benchmark.

The same methodology described previously for Twolf (Sec. 5.3) has been followed in these experiments which were carried out using the SPEC2006 reference workload. Note that, although TinySTM (not ordered) has been included for comparative purposes, it may not yield correct results. Results are shown in Fig. 17. Measured speedups and TCRs are referred to the loops under study.

The 429.mfc loop features a RAW dependence that carries all evaluated STM systems to serialization. Although ReduxSTM could exploit a reduction sentence in the loop body, the use of the reduction variable as index in a subsequent array access prevent to gain any benefit (see Table 1). Extra optimizations like data forwarding could be necessary to improve the performance in this case [14].

No true data conflicts between iterations can be found in the 433.milc loop. In this case the STM system scalability is mainly limited by false positives, especially for TinySTM which exhibits a very low TCR. The lack of data locality in STM ver-

Table 4: STAMP suite arguments and features.

Bench	Parameters	#xact	xact length	R/W Set	xact time	Contention
Bayes	-v32 -r4096 -n10 -p40 -i2 -e8 -s1	2518	Long	Large	High	High
Genome	-g16384 -s64 -n16777216	2139692	Medium	Medium	High	Low
Intruder	-a10 -l128 -n262144 -s1	23428126	Short	Medium	Medium	High
KMeans	-m40 -n40 -t0.00001 -i random-n65536-d32-c16	87382	Short	Small	Low	Low
Labyrinth	-i random-x512-y512-z7-n512	1026	Long	Large	High	High
SSCA2	-s20 -i1.0 -u1.0 -l3 -p3	22362279	Short	Small	Low	Low
Vacation	-n2 -q90 -u98 -r1048576 -t4194304	4194304	Medium	Medium	High	Low/Medium
Yada	-a15 -i ttimeu1000000.2	2415298	Long	Large	High	Medium

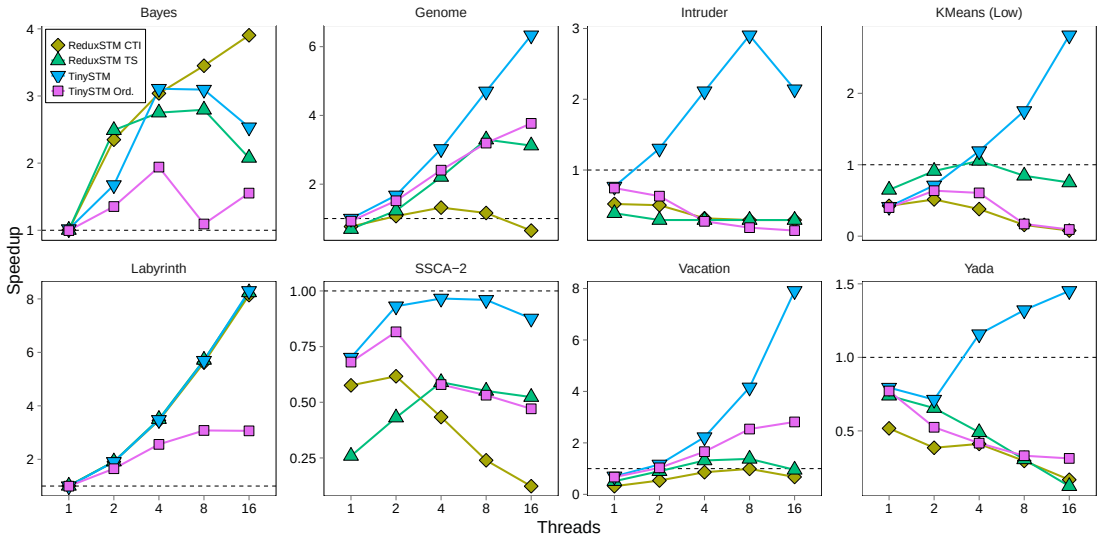


Figure 15: STM comparison in STAMP suite.

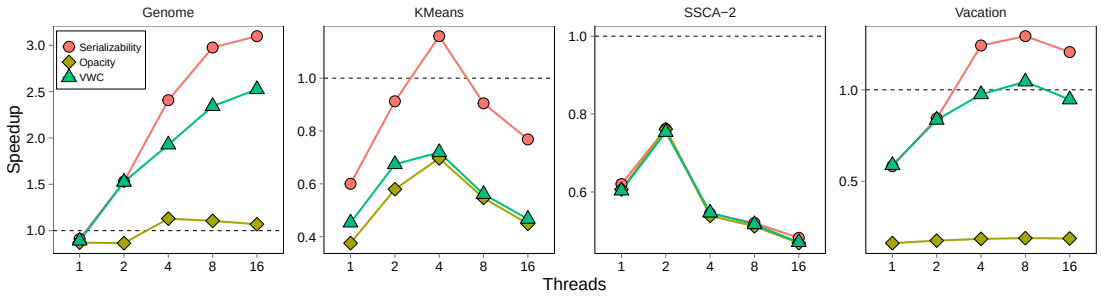


Figure 16: ReduxSTM-TS speedup with different consistency constraints for those STAMP codes that can run with relaxed consistency.

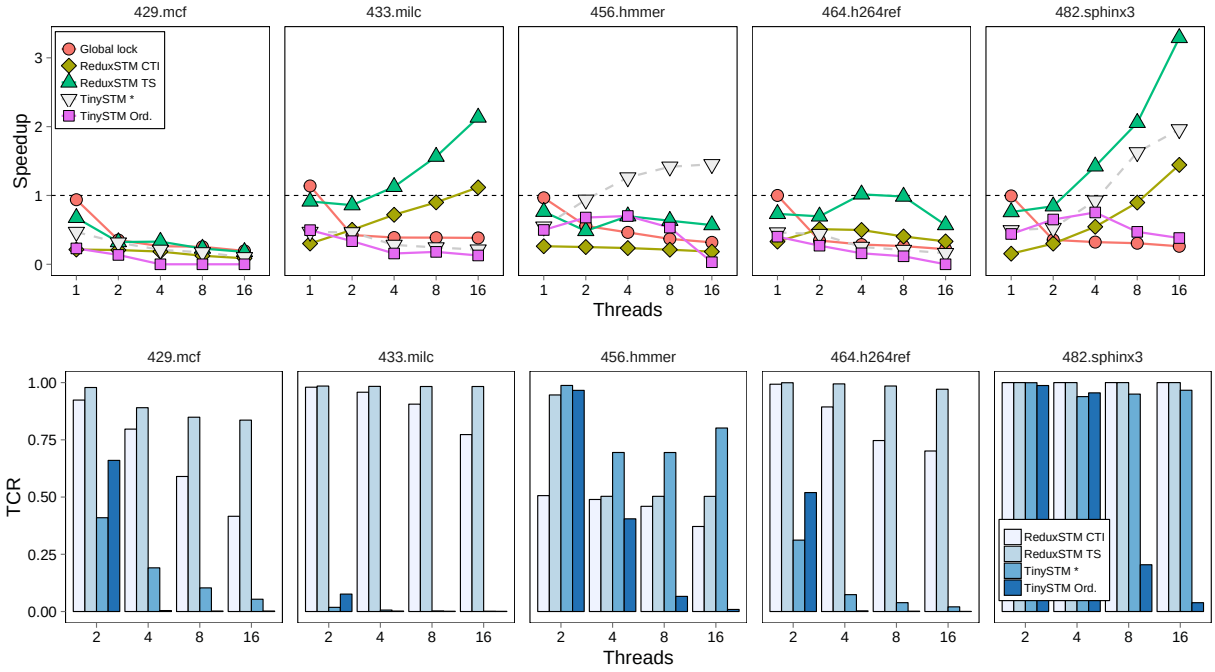
sions with respect the sequential one also limits their performance.

The 456.hmmer loop includes a recurrence that makes the current iteration depend on the previous one (RAW dependence). This causes the serialization of transactions in ordered STM systems unless optimizations like data forwarding are available. The higher TCR observed in not ordered TinySTM is explained by those conflicts that do not emerge when sequential order is not fulfilled (results of this version may be incorrect).

Potential WAW dependences can appear in the 456.h264ref loop due to false positives. Although the ability of ReduxSTM to filter this class of conflicts gives it advantage in terms of TCR against TinySTM, the serialized commits prevent from reaching a competitive speedup because of the low computational intensity.

Conflicts in the 482.sphinx3 loop are scarce and involve WAW dependences due to false positives. In this loop both versions of ReduxSTM obtain a very high TCR, as false conflicts due to aliases in the Bloom filters or timestamp arrays correspond to WAW, which can be filtered effectively. This is not the case of TinySTM-ordered whose TCR is much lower.

Note that ReduxSTM does not suffer from some of the drawbacks cited in [14] for thread-level speculation using HTM, such as aborts caused by order inversion and WAR/WAW dependences.



*Correct results are not guaranteed for not ordered TinySTM that is shown only as a reference

Figure 17: Speedup and TCR for selected loops from SPEC CPU2006.

6. Related Work

Over the last years, STM has been subject of an extensive study that has led to an ample number of different STM algorithms. A good taxonomy can be found in [58] which discusses how this large number of available algorithms may even hinder its use, as each one exhibits advantages for some particular codes or memory patterns. ReduxSTM design, although introducing new semantics, borrows some characteristics from other STMs such as time-based STMs (TinySTM/LSA [59, 43], TL2[42]), TML [60], NOrec [49] and InvalSTM [40]. The use of TM as a means of speculative parallelization has been explored in the field of scientific computation as in [9, 61, 62] and [63] as well as the parallelization of legacy or binary code [64, 13].

As a precedent, speculative parallelization of loops like LRPD [28] relies on a inspector/executor paradigm in which iteration dependences are determined for static variables before the runtime concurrently launches groups of non-conflicting iterations. LRPD is able to privatize reduction variables suitably.

More recently similar ideas has been reformulated in *Privateer* [16], a TLS system able to support pointers and dynamic data structures. Privateer introduces a privatization criterion, that determines if a loop can be executed fully parallel, and a reduction criterion capable of deal with reduction objects. Any other cross-iteration dependence results in a violation of the the privatization criterion, so the speculative execution is discarded, and the loop is re-executed sequentially.

In the context of ordered TLS, IPOT (Implicit Parallelism With Ordered Transactions) [7] is a programming model that allows parallelizing a sequential code by defining chunks of instructions to be executed in parallel using TM-styled structures. IPOT requires additional compiler and architectural support for the speculative execution environment. IPOT features a set of explicit hint annotations for chunks and defines several groups of variables (read-only / private / reductions) that enables to relax consistency properties.

Similarly to IPOT, ALTER [65] proposes another TM-style TLS scheme. Variables can be annotated in order to use a more permissive consistency checking like *out-of-order* (TLS with no ordering) or *stale read* (ignoring read dependences). Also a variable can be declared as reduction according to a reduction operator. ALTER uses a fork-join scheme for the annotated loops, executing chunks of iterations in TM-like structures and validating results at the end of chunk.

Another TLS approach for reduction patterns is presented in [26] focused on partial reduction variables (PRV). By means of a PRV detection algorithm, speculative tasks are created, which are executed in parallel. Threads stall when a reduction variable is accessed by a not reduction operation. Conflicts between reductions are not causing stalls as they operate on private replicas that must be committed eventually. Specific architectural support is needed.

In [32] it is explored the parallelization of pure histogram reduction loops using TM as a form of selective privatization. Since only reduction operations take place in these cases, all conflict detection can be disabled during the execution of the loop whose iterations have been mapped into transactional chunks. The underlying STM system is TEPO [15], a scheduler of transactions, built on top of TinySTM, that allows transaction execution in a defined order.

Read-Modify-Write (RMW) without aborts [55] is a recent STM proposal dealing with this operation pattern (RMW) of

which reductions operations are a particular case. RMW shares with ReduxSTM some features such as declaring specific data sets, degrading reductions to writes if a memory position is updated at not reduction sentences, or deferring final accumulations to commit phase. Nevertheless, RMW does not introduce any order restriction between transactions and its scalability is strongly restricted by the number of reduction variables that is a limiting factor. The proposed algorithm may suffer also from a not negligible computation replication of the *modify* function.

Concepts behind ReduxSTM are general enough to be applicable to any given STM. In contrast to systems like IPOT or ALTER that introduce an own notation, ReduxSTM relies on standard TM explicit primitives which simplifies the programmability. Being true that RMW concept is more general than reductions themselves, ReduxSTM manages to achieve a good trade-off that allows to provide additional parallelization opportunities in irregular codes, while maintaining a reasonable level of additional instrumentation and limiting the overhead.

7. Conclusions

Transactional memory (TM) is a parallel programming paradigm suitable for being applied to irregular applications, where exploitation of optimistic concurrency could be very effective as data dependences are usually unknown until run-time. In this context, this work takes the way of improving TM by adding specific support to deal efficiently with some common memory access patterns. Specifically, we have focused on reduction patterns which are frequently found in the core of this class of applications. With this purpose, ReduxSTM, a software TM system, has been introduced. ReduxSTM combines specific support for commutative and associative operators with a mechanism for enforcing some transactional commit order, needed when reduction patterns co-exist with other memory patterns. In this way, the conflict manager is able to avoid many of the transaction aborts caused by reductions.

An extensive experimental evaluation has shown that ideas behind ReduxSTM can improve the performance of STM designs. ReduxSTM has been compared to a state-of-the-art STM system (TinySTM) for a wide set of benchmark codes, combining reduction patterns with other memory patterns in different degrees. The obtained results encourage to include this type of support in STM systems.

Acknowledgements

This work has been supported by the Government of Spain with project TIN2013-42253-P.

References

[1] M. Herlihy, J. Moss, Transactional memory: Architectural support for lock-free data structures, in: 20th Ann. Int'l. Symp. on Computer Architecture (ISCA'93), 1993, pp. 289–300.

[2] T. Harris, J. Larus, R. Rajwar, Transactional Memory, 2nd Ed., Morgan & Claypool Publishers, USA, 2010.

[3] J. Larus, R. Rajwar, Transactional Memory, Morgan & Claypool Publishers, USA, 2007.

[4] R. M. Yoo, C. J. Hughes, K. Lai, R. Rajwar, Performance evaluation of Intel transactional synchronization extensions for high-performance computing, in: Int'l. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'13), ACM Press, 2013, pp. 1–11.

[5] M. Ohmacht, A. Wang, T. Gooding, B. Nathanson, I. Nair, G. Janssen, M. Schaal, B. Steinmacher-Burow, IBM Blue Gene/Q memory subsystem with speculative execution and transactional memory, IBM J. on Research and Development 57 (1/2) (2013) 7:1–12.

[6] H. Le, G. Guthrie, D. Williams, M. Michael, B. Frey, W. Starke, C. May, R. Odaira, T. Nakaike, Transactional memory support in the IBM POWER8 processor, IBM J. on Research and Development 59 (1) (2015) 8:1–14.

[7] C. von Praun, C. Ceze, C. Cascaval, Implicit parallelism with ordered transactions, in: 12th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'07), 2007, pp. 79–89.

[8] L. Porter, B. Choi, D. M. Tullsen, Mapping out a path from Hardware Transactional Memory to Speculative Multithreading, in: 18th Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'09), 2009, pp. 313–324.

[9] K. Nikas, N. Anastopoulos, G. Goumas, N. Koziris, Employing transactional memory and helper threads to speedup Dijkstra's algorithm, in: 38th Int'l Conf. on Parallel Processing (ICPP'09), 2009, pp. 388–395.

[10] M. Mehrara, J. Hao, P.-C. Hsu, S. Mahlke, Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory, in: 30th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'09), 2009, pp. 166–176.

[11] C. E. Oancea, A. Mycroft, T. Harris, A lightweight in-place implementation for software thread-level speculation, in: 21st Annual Symp. on Parallelism in Algorithms and Architectures (SPAA'09), 2009, pp. 223–232.

[12] J. Barreto, P. F. Dragojevic, R. Filipe, R. Guerraoui, Unifying thread-level speculation and transactional memory, in: ACM/IFIP/USENIX 13th Int'l. Middleware Conf. (Middleware'12), 2012, pp. 187–207.

[13] M. Saad, M. Mohamedin, B. Ravindran, HydraVM: extracting parallelism from legacy sequential code using STM, in: 4th USENIX Workshop on Hot Topics in Parallelism (HotPar'12), 2012.

[14] R. Odaira, , T. Nakaike, Thread-level speculation on off-the-self hardware transactional memory, in: IEEE Int'l. Symp. on Workload Characterization (IISWC'14), 2014, pp. 212–221.

[15] M. A. Gonzalez-Mesa, E. Gutierrez, E. L. Zapata, O. Plata, Effective transactional memory execution management for improved concurrency, ACM Transactions on Architecture and Code Optimization 11 (3) (2014) 24.

[16] N. P. Johnson, H. Kim, P. Prabhu, A. Zaks, D. I. August, Speculative separation for privatization and reductions, in: 33rd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'12), 2012, pp. 359–370.

[17] M. Schindewolf, A. Cohen, W. Karl, A. Marongiu, L. Benini, Towards transactional memory support for gcc, in: GCC Research Opportunities Workshop, 2009.

[18] H. Yu, L. Rauchwerger, An adaptive algorithm selection framework for reduction parallelization, IEEE Trans. on Parallel and Distributed Systems 17 (10) (2006) 1084–1096.

[19] R. Jin, G. Yang, G. Agrawal, Shared memory parallelization of data mining algorithms: Techniques, programming interface, and performance, IEEE Trans. on Knowledge and Data Engineering 17 (1) (2005) 71–89.

[20] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bu, Maximizing multiprocessor performance with the SUIF compiler, IEEE Computer 29 (12) (1996) 84–89.

[21] P. Feautrier, Array expansion, in: 2nd Int'l Conf. on Supercomputing (ICS'88), 1988, pp. 429–441.

[22] H. Han, C. Tseng, Exploiting locality for irregular scientific codes, IEEE Trans. on Parallel and Distributed Systems 17 (7) (2006) 606–618.

[23] E. Gutiérrez, O. Plata, E. Zapata, A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors, in: 14th Int. Conf. on Supercomputing (ICS'00), 2000, pp. 78–87.

[24] L. Rauchwerger, Speculative parallelization of loops, in: Encyclopedia of Parallel Computing, Springer, 2011, pp. 1901–1912.

[25] Systems performance evaluation cooperation. SPEC benchmarks, `http://www.spec.org`, retrieved: 2015-07-19.

[26] L. Han, W. Liu, J. M. Tuck, Speculative parallelization of partial reduction

variables, in: 8th Annual IEEE/ACM Int'l. Symp. on Code Generation and Optimization (CGO'10), 2010, pp. 141–150.
[27] C. Cao Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: Stanford transactional applications for multi-processing, in: IEEE Int'l. Symp. on Workload Characterization (IISWC'08), 2008, pp. 35–46.
[28] L. Rauchwerger, D. Padua, The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization, IEEE Transactions on Parallel and Distributed Systems 10 (2) (1999) 160–180.
[29] V. Ravi, G. Agrawal, Integrating and optimizing transactional memory in a data mining middleware, in: Int'l. Conf. on High Performance Computing (HiPC'09), 2009, pp. 215–224.
[30] S. Kim, H. Han, K.-M. Choe, Region-based parallelization of irregular reductions on explicitly managed memory hierarchies, The Journal of Supercomputing 56 (1) (2011) 25–55.
[31] W. Baek, C. Minh, M. Trautmann, C. Kozyrakis, K. Olukotun, The OpenTM transactional application programming interface, in: 16th Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'07), 2007, pp. 376–387.
[32] M. Gonzalez-Mesa, R. Quislant, E. Gutierrez, O. Plata, Dealing with reduction operations using transactional memory, in: 25th Int'l. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD'13), 2013, pp. 128–135.
[33] M. L. Scott, Sequential specification of transactional memory semantics, in: 1st ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'06), 2006.
[34] P. Fatourou, M. Iaremko, E. Kanellou, E. Kosmas, Algorithmic techniques in STM design, in: Transactional Memory. Foundations, Algorithms, Tools, and Applications, Springer, 2015, pp. 101–126.
[35] M. F. Spear, V. J. Marathe, W. N. Scherer III, M. L. Scott, Conflict detection and validation strategies for software transactional memory, in: 20th Int'l. Symp. on Distributed Computing (DISC'06), 2006, pp. 179–193.
[36] H. E. Ramadan, I. Roy, M. Herlihy, E. Witchel, Committing conflicting transactions in an STM, in: 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'09), 2009, pp. 163–172.
[37] M. F. Spear, L. Dalessandro, V. J. Marathe, M. L. Scott, A comprehensive strategy for contention management in software transactional memory, in: 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09), 2009, pp. 141–150.
[38] R. Guerraoui, M. Kapalka, On the correctness of transactional memory, in: 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'08), 2008, pp. 175–184.
[39] D. Dziuma, P. Fatourou, E. Kanellou, Consistency for transactional memory computing, in: Transactional Memory. Foundations, Algorithms, Tools, and Applications, Springer, 2015, pp. 3–31.
[40] J. E. Gottschlich, M. Vachharajani, J. G. Siek, An efficient software transactional memory using commit-time invalidation, in: 8th Ann. IEEE/ACM Int'l. Symp. on Code Generation and Optimization (CGO'10), 2010, pp. 101–110.
[41] D. Imbs, M. Raynal, Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations), Theoretical Computer Science 444 (2012) 113–127.
[42] D. Dice, O. Shalev, N. Shavit, Transactional locking ii, in: 20th Int'l. Conf. on Distributed Computing (DISC'06), 2006, pp. 194–208.
[43] P. Felber, C. Fetzer, P. Marlier, T. Riegel, Time-based software transactional memory, IEEE Trans. on Parallel and Distributed Systems 21 (12) (2010) 1793–1807.
[44] M. F. Spear, M. M. Michael, M. L. Scott, P. Wu, Reducing memory ordering overheads in software transactional memory, in: 7th Annual IEEE/ACM Int'l. Symp. on Code Generation and Optimization (CGO'09), 2009, pp. 13–24.
[45] L. Ceze, J. Tuck, J. Torrellas, C. Cascaval, Bulk disambiguation of speculative threads in multiprocessors, ACM SIGARCH Computer Architecture News 34 (2) (2006) 227–238.
[46] M. F. Spear, K. Kelsey, T. Bai, L. Dalessandro, M. L. Scott, C. Ding, P. Wu, Fastpath speculative parallelization, in: Int'l. Workshop on Languages and Compilers for Parallel Computing (LCPC'09), 2009, pp. 338–352.
[47] R. Quislant, E. Gutierrez, O. Plata, E. L. Zapata, Hardware signature designs to deal with asymmetry in transactional data sets, IEEE Transactions on Parallel and Distributed Systems 24 (3) (2013) 506–519.
[48] C. Lameter, Effective synchronization on Linux/NUMA systems, in: Gelato Conference, 2005.
[49] L. Dalessandro, M. F. Spear, M. L. Scott, NOrec: Streamlining STM by abolishing ownership records, in: 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'10), 2010, pp. 67–78.
[50] A. Fog, Instructions for ASMlib: a multiplatform library of highly optimized functions for C and C++, Technical University of Denmark, version 2.34 (2013).
[51] L. Dalessandro, D. Dice, M. Scott, N. Shavit, M. Spear, Transactional mutex locks, in: 16th Int'l. Euro-Par Conf. (EuroPar'10), 2010, pp. 2–13.
[52] T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, K. Olukotun, Eigenbench: A simple exploration tool for orthogonal TM characteristics, in: IEEE Int'l. Symp. on Workload Characterization (IISWC'10), 2010, pp. 1–11.
[53] F. Zyulkyarov, A. Cristal, S. Cvijic, E. Ayguade, M. Valero, O. Unsal, T. Harris, WormBench - a configurable workload for evaluating transactional memory systems, in: 17th Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'08), 2008, pp. 61–68.
[54] R. Das, Y.-S. Hwang, J. Saltz, A. Sussman, Runtime and compiler support for irregular computations, in: Compiler optimizations for scalable parallel systems, Springer, 2001, pp. 751–778.
[55] W. Ruan, Y. Liu, M. Spear, Transactional read-modify-write without aborts, ACM Transactions on Architecture and Code Optimization 11 (4) (2015) 1–24.
[56] W. Ruan, Y. Liu, M. Spear, STAMP need not be considered harmful, in: 9th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'14), 2014.
[57] V. Packirisamy, A. Zhai, W. C. Hsu, P. C. Yew, T. F. Ngai, Exploring speculative parallelism in SPEC2006, in: ISPASS, 2009, pp. 77–88. doi:10.1109/ISPASS.2009.4919640.
[58] Q. Wang, S. Kulkarni, J. Cavazos, M. Spear, A transactional memory with automatic performance tuning, ACM Transactions on Architecture and Code Optimization 8 (4) (2012) 1–23.
[59] P. Felber, C. Fetzer, T. Riegel, Dynamic performance tuning of word-based software transactional memory, in: 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'08), 2008, pp. 237–246.
[60] M. F. Spear, A. Shriraman, L. Dalessandro, M. L. Scott, Transactional mutex locks, in: 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'14), 2009.
[61] K. Ljungkvist, M. Tillenius, D. Black-Schaffer, S. Holmgren, M. Karlsson, E. Larsson, Using hardware transactional memory for high-performance computing, in: IEEE Int'l. Symp. on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW'11), 2011, pp. 1660–1667.
[62] J. Sreeram, S. Pande, Parallelizing a real-time physics engine using transactional memory, in: 17th Int'l. Euro-Par Conference (Euro-Par'11), 2011, pp. 206–223.
[63] B. L. Bihari, Transactional memory for unstructured mesh simulations, Journal of Scientific Computing 54 (2-3) (2013) 311–332.
[64] M. DeVuyst, D. M. Tullsen, S. W. Kim, Runtime parallelization of legacy code on a transactional memory system, in: 6th Int'l. Conf. on High Performance and Embedded Architectures and Compilers (HiPEAC'11), 2011, pp. 127–136.
[65] A. Udupa, K. Rajan, W. Thies, ALTER: Exploiting breakable dependences for parallelization, in: 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'11), 2011, pp. 480–491.